Java Effective Second Edition.

1. Introduction.

Saya yang meringkas dan menulis ulang materi tentang Java Effective 2nd Edition ini bernama Yuhariz Aldyan. Tidak ada tujuan lain, hanya belajar dan mendokumentasikan kembali dengan bahasa Indonesia, agar memudahkan saya untuk membaca ulang seandainya diperlukan suatu saat nanti. Tidak ada tujuan apapun selain itu, seandainya pun materi ini akan di upload, tujuannya hanya untuk berbagi seandainya ada yang perlu materi untuk memperluas ilmu tentang java mereka. Tidak ada tujuan komersil ataupun membanggakan diri. Semoga materi ini berguna dan ilmunya bisa bermanfaat bagi siapa saja yang membacanya. Terima Kasih.

Source code serta sedikit penjelasan bisa di-akses : *https://github.com/YAldyan/Java-Effective*

2. Creating And Destroying Object.

Sekarang kita fokus pada bagaimana objek diciptakan dan lalu dihancurkan, spesifik sampai tingkat apa proses penghancuran objek itu, objek itu harus benar-benar bersih.

Item 1. Pertimbangkan menggunakan static factory method daripada constructor.

static method factory adalah sebuah method yang mengembalikan instance dari objek tertentu. Contohnya seperti berikan di bawah ini.

```
public static boolean valueOf(boolean  b){
        return b ? Boolean.TRUE : Boolean.FALSE ;
}
```

static factory method berbeda dengan *factory method* di *design pattern, mereka* tidak ada kaitan secara langsung antar satu sama lain. Ada

beberapa keunggulan *static factory method* daripada *constructor*, yaitu dibawah ini.

1. Static factory method punya nama, sedangconstructor tidak. Maksudnya adalah umumnya *constructor* tidak ada nama, artinya hanya *new()* kelas tertentu dan jadilah objek itu. Bandingkan antara *constructor* untuk bilangan prima pada *BigInteger, BigInteger(int, int, Random)* yang akan menghasilkan nilai dengan kemungkinan bilangan prima. Dengan *static factory method, akan* menjadi *BigInteger.probablePrime(int).* Hanya menggunakan 1 parameter dan punya nama yang spesifik.

kasus umum berikutnya untuk contoh langkah ini, biasanya kita akan membuat 2 *constructor* jikalau ada lebih dari satu instance yang akan aktif dan jalan, dan umumnya antara satu dan yang lain kita bedakan menggunakan parameter, ada constructor yang satu parameter, dua parameter, bahkan tidak ada parameter sama sekali. Ini adalah pilihan yang buruk, sebab user sulit untuk ingat tujuan tiap2 *constructor* persisnya buat apa, kalo ada kejadian begini,yang melibatkan banyak *constructor* dengan tujuan berbeda, maka cara terbaik adalah menggunakan *static factory method,* agar lebih muda mengenalinya, karena mereka punya nama, seperti Contoh di atas terkait BigInteger prima, kita bisa buat satu lagi *static factory method* untuk bilangan kelipatan 5 misalnya, dengan *probable5X().* Itu akan sangat dikenali dari namanya, terlihat lebih baik dari *constructor.*

2. Tidak perlu create new objek tiap kali di eksekusi. Maksudnya, proses create objek akan bisa kita kontrol, jika kita perlu objek yang sama, tidak perlu lagi untuk create ulang, tinggal kita re-invoke aja, ini mencegah objek ganda, dan memastikan objek itu singleton. Sehingga secara tidak langsung akan meningkatkan performance.

3. Tidak seperti *constructor, static factory method* bisa mengembalikan banyak jenis objek, yang merupakan turunan dari objek yang seharusnya di-return.

4. Bisa membuang redudansi parameter ketika melakukan instances terhadap objek tertentu.

```
Map<String, List<String>()> map = new HashMap<String, List<String>()>();

change to correct,

Map<String, List<String>()> map = HashMap.newInstances();
```

Kelemahan static factory method antara lain,

1. Method dengan *accessor* selain *public* dan *protected* nggak bisa di *subclass.* Itu berarti lebih baik *composition* daripada *inheritance.*

Item 2. Pertimbangkan menggunakan *builder* jikalau bertemu dengan *constructor* dengan banyak parameter.

sebenarnya, salah satu cara yang bisa di pake buat mengatasi *multiple parameter* di*constructor* adalah dengan *Java beans,* dengan menggunakan method *setter to set all the value and assign it* otomatis dengan *public constructor.* Tapi, terdapat cukup banyak kelemahan, yaitu kita tidak bisa mengecek validitas data yang di passing ke setter dan diteruskan untuk create objek. masalah berikutnya yang juga sulit adalah *Java* beans punyakemungkinan membuat datanya *immutable,* dan itu akan sangat sulit untuk maintenance codenya.

akhirnya, cara yang paling berguna adalah dengan menggunakan *builder pattern,* bagian Dari *design pattern*. Contoh kodenya dibawah ini.

```
public NutritionFacts{

    private int serving size ;

    private int serving;

    private int lemak;

    private int kalori;
```

```
private int sodium;
private int karbohidrat;

public static class Builder{
        private final int servingSize;
        private final int serving;

        private int lemak;
        private int kalori;
        private int sodium;
        private int karbohidrat;
        public Builder(int servingSize, int servings){
                this.servingSize = servingSize;
                this.servings = servings;
        }
        public Builder lemak(int val){
                this.lemak = val;
                return this;
        }
        public Builder kalorI(int val){
                this.kalori = val;
                return this;
        }
        public Builder sodium(int sodium){
```

```
            this.sodium = sodium ;
            return this ;
        }
        public Builder karbohidrat(int val){
            this.karbohidrat = val ;
            return this ;
        }
        public NutritionFacts build(){
            return new NutritionFacts(this);
        }
    }
    private NutritionFacts(Builder builder){
        servingSize = builder.servingSize;
        serving = builder.serving;
        lemak = builder.lemak;
        kalori = builder.kalori;
        sodium = builder.sodium;
        karbohidrat = builder.karbohidrat;
    }
}
```

NutritionFacts bersifat *immutable,* artinya nilainya tetap tidak bisa dimodifikasi dengan mengganti menjadi nilai baru. Cara eksekusi *builder pattern* di atas, seperti di bawah ini.

```
NutritionFacts nf = new NutritionFacts.builder(2,3).lemak(4).kalori(9).sodium(8).karbohidrat(10);
```

builder pattern menjadikan kode mudah untuk ditulis dan tentunya dibaca, builder pattern mensimulasikan nama-nama parameter.

Builder parameter sebaliknya digunakan pada kelas yang membutuhkan 4 parameter atau lebih untuk proses penciptaannya, kita juga harus memikirkan kemungkinan sebuah kelas akan menambah parameter dalam proses penciptaannya, maksudnya adalah ketika kelas itu punya potensi untuk memiliki 4 parameter atau lebih, lebih baik sedari awal kita membuat *builder-nya,* karena ketika kita akan membuat*builder* sementara kelas udah tercipta dan jalan tampa *builder,* maka itu adalah pilihan yang buruk.

Item 3. Usahakan untuk menggunakan singleton property dan private constructor atau enum type.

Singleton adalah objek yang hanya di create sekali saja. Tidak akan ada objek yang sama tipenya diciptakan kemudian. Contoh codenya, dibawah ini.

```
// singleton static factory method.

public class Rumah{

    private static final Rumah instance = new Rumah();

    private Rumah(){};

    public Rumah getInstance{ return instance; }

}
```

terdapat kelemahan pada *singleton* dengan format diatas, jikalau kita butuh kelas *serializable,* maka kelas di atas harus *implement serializable*, itu mengakibatkan instance tunggal kita mesti ditambahkan "*transient*" didepan type ketika deklarasi variable, agar tidak terkena dampak *serializable,* karena dampaknya akan mencetak objek itu berkali- kali *serial serializable.*

```
// enum singleton - format yang lebih baik untuk singleton.

public enum Rumah{

        INSTANCE;

        public void leaveBuilding{....};

}
```

enum singleton adalah cara terbaik untuk menerapkan *singleton.*

Item 4. Pastikan kelas tidak bisa diinstansiasi dengan accessor private pada constructornya.

Untuk mencegah instance sebuah kelas agar tidak di *invoke* dari luar kelas itu, kita bisa membuatnya *non-instance* dengan *private constructor.* Ada beberapa kelas, atau kelompok kelas di Java yang menerapkan ini, misalnya *Arrays* dan *Math,* mereka tidak bisa diinstansiasi meskipun oleh *subclassnya,* karena *private constructor,* sehingga mencegah instansiasi yang seenaknya.

Item 5. Jangan menciptakan objek ketika kamu bisa reuse objek yang sudah exist.

Hindarkan untuk membuat code yang akan menyebabkan proses instansiasi berulang, misalnya hindarkan untuk melakukan new di method yang akan sering di *invoke,* sebab tiap kali di *invoke* mereka akan menciptakan objek baru, dan itu membuat program berjalan lebih lambat. Karena terlalu banyak objek yang sia-sia akibat creating objek terlalu berlebihan.

Item 6. Buang objek reference yang tidak terpakai.

Ketika objek sudah tidak terpakai lagi, maka harus dipastikan objek yang menjadi referencenya harus di null-kan, atau dihapus.

Yang menjadi masalah berikutnya adalah *cache, cache* ini memiliki problem yang mirip dengan sebelumnya ialah menyebabkan *memory leak.* Ketika kamu meletakkan objek reference di sana, cukup mudah untuk melupakan

itu, dan tetap membiarkannya di sana saat objeknya sudah tidak *relevant* lagi, atau *irrelevant.*

Berikutnya adalah *register listener* dan *callback,* ketika API mendaftarkan *listener* dan *callback* tapi tidak di *deregister,* maka mereka akan di akumulasikan, kecuali anda melakukan action tertentu.

Item 7. Hindari menggunakan finalizer.

Menghindari penggunaan *finalize* karena *unpredictable,* sering membahayakan, dan umumnya tidak berguna.

Kapan *finally Java* sebaiknya digunakan**.**

1. Ketika akan *terminate resource* tertentu, biasanya terkait dengan, *FileInputStream, FileOutputStream, Timer, Connection,* mungkin kalo di Java bisa dikelompokkan berdasarkan *library, Java Input Output, Java SQL Connection,* dan *Java Thread.*

2. Native Peer, native kalo tidak salah adalah memanggil objek Java dengan objek lain, misalnya objek *C++,* atau sebaliknya, karena ini merupakan objek yang jenisnya *special,* maka *Garbage Collector* tidak bisa menghandle ini, sehingga kita mesti membebaskan sendiri atau *terminate* sendiri dengan *try-finally* di Java*.*

3. Method Common To All Objects.

Terdapat beberapa object yang akan diturunkan pada setiap kelas yang diciptakan, asalkan kelas itu *non-final,* sehingga memungkinkan untuk menjadi *superclass*, karena juga sebenarnya kelas yang diciptakan itu merupakan *subclass* dari *Class object.* Beberapa method bawakan, yaitu *equals(), hascode(), toString(), clone(), finalize(),* Terakhir *Comparable.compareTo(),* sebenarnya sedikit berbeda, tapi ini juga dibahas di chapter ini.

Item 8. Mematuhi Aturan Umum Ketika Override equals().

Aturan untuk tidak meng-overrides equals().

1. Tiap kelas yang diciptakan itu bersifat unik.

2. Kamu yakin kelas itu tidak butuh *logical test dengan equal.*

3. Superclass sudah *override equals,* dan *superclass* merupakan bagian dari *subclass.*

4. Kelasnya *private* dan terletak di *private package,* dan diyakini *method equal* tidak akan pernah digunakan.

Aturan umum ketika meng-override method equals().

1. *Reflexive*: untuk nilai X yang tidak bernilai null, x.equals(x) must return true.
2. *Symmetric* : x.equals(y) harus sama dengan y.equals(x).
3. *Transitive* : untuk setiap x, y, z, yang tidak bukan null, jika x.equals(y) returns true dan y.equals(z) returns true, maka x.equals(z) must return true.
4. *Consistent*: jika dua buah objek adalah sama, mereka harus selalu sama dalam selamanya, kecuali salah satu atau kedua objek itu di modifikasi.
5. *Non-Nullity* : jika sebuah nilai x bukan null, x.equals(null)haruslah bernilai false.

High Quality ***equals() method*****.**

1. Use == operator to check apakah me reference objek yang sama, atau bertempat di alamat memory yang sama.

Contoh, ***String nama == String name ;***

2. Use "instanceof" operator to check tipe data dari objek.

Contoh, ***O instanceof Objek*****.**

3. Casting variabel ke tipe data atau jenis objek tertentu.

Contohnya, ***int nilai = (int) desimal ;***

4. Method .equals(), untuk membandingkan nilai field atau variabel objek satu dengan yang lainnya.

Contoh, ***field.equals(o.field2).***

Item 9. Setiap .equals() di override, maka .hascode() akan harus di override juga.

Aturan untuk *.hascode ().*

1. Ketika di eksekusi berkali-kali, *hascode* tetap memberikan nilai *integer* yang sama*.*

2. Jika *.equals()* memberikan nilai *true,* maka kedua variabel yang dibandingkan harus mempunyai nilai *hascode* yang sama. Tetapi tidak berlaku sebaliknya*.*

3. Jika *.equals()* memberikan nilai *false,* maka nilai *hascode* yang diperoleh antar dua field yang dibandingkan harus berbeda.

Jika satu ketika kita mempunyai objek, lalu objek itu di *insert* ke dalam sebuah *collection,* katakanlah *HashMap,* di kelas yang merupakan tipe instance dari objek itu, method *.equals()* di *override,* sedangkan *.hascode()* tidak, maka ketika kita akan *.get()* nilai dariCollection itu, dia akan mengembalikan nilai *NULL,* karena apa ? karena sebelum proses *.equals()* dari objek itu dipanggil, dia akan memanggil *.hascode()* dulu, dan ternyata dia tidak menemukan *hascode,* karena tidak di *override,* sehingga dia tidak mengembalikan nilai, dan mengembalikan *NULL value.*

Kesimpulannya, ketika kita create object, maka object itu membawa nilai *hascode* bersama dengan dirinya, dan ketika proses *insert* ke *collection,* nilai *hascode* itu juga dibawa ke dalam *collection* tersebut. Tahap selanjutnya

adalah ketika objek itu akan di-retrieve dari *collection,* maka kita atau program menciptakan objek baru untuk dibandingkan dengan yang tersimpan di *collection,* objek baru ini juga membawa kode *hascode* ketika diciptakan*,* untuk objek yang sama dengan variabel yang sama yang dimilikinya, maka dia akan menghasilkan *hascode* yang sama pula, berapa kalipun diexecute, *hascode* yang sama akan dihasilkan. Kenapa hascode terlebih dahulu dibandingkan ketimbang *.equals(),* padahal logikanya kita bisa membandingkan secara langsung. Ada teori tentang *hascode* dan *equals* yang khusus, tentang sifat antara mereka*.*

Teori 1. Kalau *.equals()* sama, maka *.hascode()* yang dihasilkan atau dipunyai pasti sama*.*

Teori 2. Kalau *.hascode()* sama, belum pasti *.equals()* sama.

Akhirnya kita bisa menyimpulkan bahwa, sangat benar untuk membandingkan *hascode* dulu baru kemudian *equals.* Kalau hascode saja sudah tidak sama, tidak perlu buang-buang waktu untuk memastikan dengan *equals,* karena hasilnya pasti *equals*-nyajuga berbeda. Sedangkan kalau *hascode sama,* mungkin saja *equals*-nya berbeda, kita perlu memastikan dengan *checking .equals().*

Cara yang benar untuk *override hascode* bisa kita baca di internet, karena terlalu matematis, jadi agak sulit buat dijelaskan dengan kata-kata, tapi kita juga dapat untuk *override* itu secara otomatis dengan fitur di IDE semacam eclipse dan netbeans.

Item 10. Selalu override .toString().

Awalnya *.toString()* seperti tidak berguna secara structural, tapi sebenarnya sangat dianjurkan untuk di *override,* alasannya adalah untuk memberitahukan *format* dari data objek yang tepat untuk memudahkan *user* untuk mengetahui secara informatif. Misalnya untuk Objek *PhoneNumber, format* yang diinginkan adalah *(021) 378066.* Untuk *.toString()* kita bisa

memakai,***return "("+areaCode+")"+number ;***. Sehingga akan memudahkan programmer untuk menentukan juga, format seperti apa yang sesungguhnya diinginkan oleh *user,* agar bersifat mudah dibaca dan informatif. Sedangkan Kalau tidak di *override,* dia akan memberikan nilai, *PhoneNumber+hashcode.* Jadi return oleh *.toString()* akan membentuk *format* yang aneh dan tidak dimengerti oleh user.

11. Override .clone() Dengan Bijaksana.

Method ini adalah method yang digunakan untuk mengcopy semua data yang ada di objek yang menggunakannya. Untuk memperoleh method ini, kita terlebih dahulu mesti *implement Cloneable interface.* Yang memiliki method tunggal *.clone().* Ada aturan yang mesti dipenuhi oleh pengguna method ini, karena kasus ini special, dia bisa menciptakan objek tanpa memanggil *constructor.* Meskipun objek yang diciptakan sama persis dengan yang menggunakan-nya, tapi tetap saja objek diciptakan tanpa memanggil *constructor.* Kondisinya seperti di bawah ini.

1. x.clone() != x.

2. x.clone().getClass() = x.getClass ()

3. x.clone().equals(x).

Aturan.

1. Kalau *clone* dilakukan pada kelas yang tidak *final,* maka untuk memanggil instance dari objek itu harus dengan *invoke super.clone().* Jika field dari objek yang akan di *clone* itu adalah primitif *field,* seperti *integer, string,* float dkk, yang merupakan *immutable reference,* maka kita bisa menciptakan objek *instance clone* dengan cara ini. Contohnya di bawah ini.

```
@override

public PhoneNumber clone(){
```

```
        try{
            return (PhoneNumber) super.clone();
        } catch(CloneNotSupportedException e){}
        return null;
}
```

2. Berikutnya, seandainya *clone* dilakukan pada objek yang fieldnya *Mutable,* maka akan terjadi masalah dengan cara di atas. Misalnya dengan kelas *Stack,* kelas *Stack* bisa dimanipulasi dengan menambah jumlah elements, maupun menguranginya, jadi akan menyebabkan masalah pada data yang tidak konsisten, kenapa begitu karena antara *clone* dan *original* sama2 merujuk ke data *original* yang sama. Kita harus modifikasi cara *clone* itu. Contohnya di bawah ini.

```
@override.
public Stack clone (){
        try{
            Stack result = (Stack) super.clone();
            result.elements = elements.clone();
            return result;
        }catch(CloneNotSupportedException e){}
        return null;
}
```

untuk elements clone kita tidak perlu casting*,* karena cloning pada *array* akan menghasilkan balikan yang berupa array pula*.*

3. Aturan kedua tidak berlaku untuk objek yang punya field bersifat *final,* karena *clone* adalah proses *assign* nilai ke tempat yang baru. Sehingga

menyebabkan *share* nilai, sedangkan *final* tidak boleh di *share* atau diturunkan ke objek lain.

Item 12. Consider Implementing Comparable.

Comparable merupakan sebuah *interface* yang bisa *di*-implement oleh kelas manapun. Sesuai dengan namanya, *Comparable* adalah sebuah *interface* dengan method tunggal yaitu, *compareTo().* Tujuan method ini adalah membandingkan nilai atau value dan umumnya digunakan untuk mengurutkan sebuah *Collection,* misalnya *Array, Set,* atau sejenisnya.

```
public interface Comparable<T> {
        int compareTo(T t);
}
```

Terdapat sejumlah aturan untuk menggunakan method ini, mirip dengan menggunakan *equals().*Hasil yang didapatkan setelah proses membandingkan adalah berupa nilai *integer, -1,0,* dan *1.* Masing-masing memiliki arti, -1 artinya kurang dari, 0 artinya sama dengan, dan 1 artinya besar dari. Bisa saja memanggil sebuah *exception,* yaitu *ClassCastException,* karena objek tidak bisa dibandingkan, bisa saja berbeda jenis, atau gagal saat *di-casting* agar memiliki jenis objek yang berbeda.

Aturan umumnya antara lain.

1. Bila terdapat dua buah nilai bukan *null,* jika *x.compareTo(y)* memberikan *exception,* maka *y.compareTo(x)* harus memberikan *exception* pula.
2. *Relation* adalah *transitif,* maksudnya, jika *x.compareTo(y)* lebih besar dari 0 dan *y.compareTo(z) > 0,* maka *x.compareTo(z)* harus lebih besar dari 0 pula.
3. Jika *x.compareTo(y)* sama dengan *y.compareTo(z),* maka hasil dari *x.compareTo(z)* harus sama dengan hasil *y.compareTo(z),* berlaku untuk semua nilai *z.*
4. Terakhir, ini sangat direkomendasikan, tapi tidak terlalu dibutuhkan, *yaitu, (x.compareTo(y) == 0) == (x.equals(y)).*

Untuk membandingkan tipe data primitive, kita bisa menggunakan < dan >, khusus untuk tipe data desimal, kita bisa menggunakan*Double.compare*atau*Float.compare,* karena sering terjadi kesalahan yang tidak sesuai dengan aturan umum method *compareTo,* khusus untuk tipe data decimal.Terakhir untuk tipe data array, lakukan cara ini untuk semua nilai dalam array, jadi kita mesti membandingkan satu demi satu nilai di dalam array itu.

Untuk kelas yang memiliki banyak sekali field data, maka mulailah dari field yang paling penting dan memberikan pengaruh significant, artinya yang menjadi prioritas utama sebagai tolak uku pembanding, jika saat dibandingkan memberikan nilai 0, maka selesai sudah. Lalu teruskan ke field yang menjadi prioritas berikutnya, dan seterusnya, jika memberikan hasil 0 juga, berarti semua objek sama.

```
public int compareTo(PhoneNumber pn) {

        // Compare area codes
        if (areaCode < pn.areaCode)
                return -1;

        if (areaCode > pn.areaCode)
                return 1;

        // Area codes are equal, compare prefixes
        if (prefix < pn.prefix)
                return -1;

        if (prefix > pn.prefix)
                return 1;

        // Area codes and prefixes are equal, compare line numbers
        if (lineNumber < pn.lineNumber)
                return -1;

        if (lineNumber > pn.lineNumber)
                return 1;

        return 0; // All fields are equal
}
```

Kita bisa sederhanakan lagi menjadi.

```
public int compareTo(PhoneNumber pn) {

        // Compare area codes
```

```
        int areaCodeDiff = areaCode - pn.areaCode;

        if (areaCodeDiff != 0)
                return areaCodeDiff;

        // Area codes are equal, compare prefixes
        int prefixDiff = prefix - pn.prefix;

        if (prefixDiff != 0)
                return prefixDiff;

        // Area codes and prefixes are equal, compare line numbers
        return lineNumber - pn.lineNumber;
}
```

Cara di-atas, yang kedua tidak selalu bekerja, kita harus memastikan satu hal.

1. Pastikan nilai dari field tidak negative, atau lebih umum lagi, beda antar nilai tertinggi dan terendah kurang dari atau sama dengan : Integer.MAX_VALUE (2^{31}-1).

The reason this trick doesn't always work is that a signed 32-bit integer isn't big enough to hold the difference between two arbitrary signed 32-bit integers. If i is a large positive int and j is a large negative int, (i - j) will overflow and return a negative value. The resulting compareTo method will return incorrect results for some arguments and violate the first and second provisions of the compareTo contract. This is not a purely theoretical problem: it has caused failures in real systems. These failures can be difficult to debug, as the broken compareTo method works properly for most input values.

4. Classes And Interfaces.

Kita akan mempelajari tentang bagaimana membuat kelas yang baik dalam pemrograman, karena kelas dan anggotanya merupakan jantung dari pemrograman, kita akan belajar membuat kelas dan anggotanya agar robust, flexible, clean dan clear.

Item 13. Meminimalkan Akses ke Kelas dan Membernya.

Bahasan kali ini terkait dengan enkapsulasi, bagaimana kita harus menyembunyikan access ke kelas maupun anggotanya, dengan akses *modifier,* ada *public, private,* dan *protected.*

Ada sebuah kutipan menarik, - Buatlah kelas dan anggotanya sebisa mungkin tidak bisa di akses- Maknanya adalah pilih *access* modifier terendah untuk kelas dan anggotanya sesuai dengan fungsi dari kode yang ada tulis. Sederhananya mungkin bercerita tentang, buatlah kelas dan membernya semakin tidak bisa di akses, itu akan sangat baik untuk mengurangi resiko terhadap fungsi dari kelas itu.

Untuk kelas sendiri, hanya ada 2 jenis tipe *access modifier, package-private* dan *public,* kita harus mendeklarasikan kelas sebagai *public* jika ingin menjadi *public,* tetapi jika kita tidak mendeklarasikan *public* maka dia akan bertipe *package-private* secara default.Bedanya adalah, *public* itu bisa di akses dari *package* manapun, hanya perlu import kelas dan packagenya, tapi *package-private* hanya bisa di akses dari dalam package yang sama, tidak bisa selain itu.

Untuk yang selain kelas dan interface, terdapat4 *access modifier.*

1. private

2. package-private

3. protected

4. public

Kita harus sangat pintar untuk mendesign tipe identifier untuk tiap2 kelas di modul yang kita miliki, kita hanya perlu ingat kutipan di atas, untuk sebisa mungkin membuat kelas dan atau interface tersembunyi, tidak terlihat untuk di akses.

Jangan pernah membuat *public field* pada kelas *public,* kecuali merupakan *public static final fields* atau referensi dari *field* itu merupakan *immutable field.* Hanya cukup sesuaikan *return value* yang cocok dengan keinginan client kita, sehingga itu akan berpengaruh pada performance yang lebih baik. Intinya adalah sebisa mungkin untuk meminimalkan item atau variable

yang bisa di akses dan diubah nilainya, karena akan sangat memberatkan aplikasi dan mengakibatkan ketidakpastian nilai.

Item 14. Untuk kelas public, gunakan method untuk akses field, jangan membuat public field.

Public class sangat dilarang untuk membuat *field* dengan tipe *public,* karena itu membuang enkapsulasi yang harusnya di terapkan, data tersebut akan sangat berbahaya karena akan di akses secara liar di kelas atau modul yang berbeda *package* dengan tempatnya berasal*, public field*dibolehkan untuk kelas dengan tipe *private-package* dan atau *inner class,* karena mungkin akan membutuhkan data untuk di share dengan yang lainnya.

Item 15. Minimalkan Mutability, maksudnya usahakan kelas dan turunannya immutable alias tidak bisa dimodifikasi, sehingga client hanya memakai data yang memang sudah tersedia.

Kelas yang *immutable* adalah kelas yang instancenya tidak bisa dimodifikasi, yang umur dari objek itu bisa diketahui.

Ada 5 cara untuk *minimize mutability* dari kelas, yaitu.

1. Jangan sampai ada method yang bisa mengubah kondisi atau status dari objek, method ini lebih dikenal dengan mutator.

2. Pastikan kelas tidak bisa diturunkan, sebenarnya ini lebih kearah menjaga agar behaviour dari objek yang merupakan statenya tetap tidak berubah meski telah diturunkan, seandainya pun terpaksa harus diturunkan.

3. Buat semua *field* menjadi *final*, agar tidak bisa seenaknya dilempar dari satu thread ke thread lain, atau proses satu ke proses lain.

4. Pastikan semua *field* bertipe *acessor private.*

5. Pastikan akses yang khusus untuk *mutable fields.* Part ini adalah lebih ke arah mempunyai *setter* dan *getter* untuk tiap *field* yang bersifat *private,*

instance field maksudnya, sehingga state dari instance itu tetap sama dari awal diciptakan sampai akan dihancurkan mungkin, kita hanya mengolah data pada atribut yang menjadi bagian dari kelas atau instance object itu.

Jadi *immutable object* itu *simple,* karena statenya dari awal dicreate masih tetap sama sampai mungkin akan dihancurkan.

Selanjutnya tidak perlu sinkroninsasi ketika akan digunakan, karena state tidak pernah berubah, alias sama.

Terakhir bisa kita share secara bebas, maksudnya adalah kita tidak perlu ragu untuk nilai Dari instance bakal berubah, sehingga instance ini bisa digunakan oleh banyak objek atau modules lainnya.

Lanjutnya, kita tidak hanya bisa share instancenya saja, tapi juga bisa share internal variable dari instance itu. Maksudnya, kita bisa membuat sebuah instance atau atribut yang berbeda dengan internal dari instance yang sama.

Berikutnya, *immutable* objek akan menjaga dirinya dari pengaruh objek lainnya. Sehingga ketika anda memasukkan instance itu ke dalam sebuah *set* atau *map* atau *list,* kita tidak perlu ragu data itu akan berganti, atau hilang, atau tertukar nilainya.

Kelemahan satu satunya dari *immutable object* adalah kita mesti membuat objek sendiri2 buat nilai yang berbeda, meski berbeda hanya sedikit saja, dan itu akan sangat buruk untuk coding dalam jumlah raksasa, karena akan memakan cost yang banyak, loading programming dan sangat sulit buat maintenance dari code, karena mesti di sisir satu demi satu.

Ada cara yang baik untuk membuat sebuah *instance object* menjadi *immutable,* selain dengan menambahkan *final type* untuk objek itu, yang memastikan dia tidak dapat di *subclassing.* Yaitu dengan *method static factory,* contohnya dibawa ini.

public static Complex getValue(int a, int b){

```
    new Complex(a,b);
}
```

Item 16. Favor Composition over inheritance.

Inheritance tidak selalu baik, terkadang *inheritance* merupakan sebuah langkah yang salah, contohnya adalah ketika kamu menurunkan sebuah kelas, baik dalam bentuk *extends,* maupun *implements,* itu berarti kelas anak merupakan wujud dan ditentukan pula oleh skenario dari induk, bukan tidak mungkin induk akan mengalami perubahan dalam struktur kelasnya ataupun behaviour dari class itu.

Inheritance sangat baik sebagai cara untuk *reuse code,* tapi tidak selalu menjadi cara terbaik untuk tiap kondisi. Menggunakan itu secara tidak tepat akan membuat aplikasi menjadi rapuh. Sangat baik untuk menggunakan *inheritance* dalam satu *package,* yang mana antara *superclass* dan *subclass* masih berada di control oleh programmer yang sama.

Masalah yang di-diskusikan dalam topic kali ini adalah untuk tidak menerapkan *interface inheritance,* ketika sebuah kelas *meng-implement interface* atau ketika sebuah *interface meng-extends* kelas lainnya.

Untuk memilih menggunakan *inheritance.* kita harus benar2 bertanya pada diri sendiri, apakah setiap kelas dari B yang *mengextends A* adalah A, jikalau kita tidak yakin sebaiknya hindari untuk menggunakannya.

Inheritance sebenarnya sangat powerful, tapi bermasalah karena melanggar enkapsulasi.

```
// Broken - Inappropriate use of inheritance!
public class InstrumentedHashSet<E> extends HashSet<E> {
        // The number of attempted element insertions
        private int addCount = 0;
        public InstrumentedHashSet() {
```

```
    }
    public InstrumentedHashSet(int initCap, float loadFactor) {
        super(initCap, loadFactor);
    }
    @Override
    public boolean add(E e) {
        addCount++;
        return super.add(e);
    }
    @Override
    public boolean addAll(Collection<? extends E> c) {
        addCount += c.size();
        return super.addAll(c);
    }
    public int getAddCount() {
        return addCount;
    }
}
InstrumentedHashSet<String> s = new InstrumentedHashSet<String>();
s.addAll(Arrays.asList("Snap", "Crackle", "Pop"));
```

Kalau kita running program di-atas dan *execute method getAddCount,* kita berpikir bahwa akan return 3 sebagai jumlah, sebagaimana kita insert tiga item ke Set, tapi nyatanya nilai yang dikembalikan adalah 6, apakah yang salah, Mari kita pikirkan, kenyataannya adalah ketika dia telah menambahkan 3 data baru ke *Set,* kemudian dia *meng-execute* lagi dengan memanggil *super.addAll().* Ini kembali memanggil *method add,* sebagaimana telah *di-override* di kelas *InstrumentedHashSet* masing-masing satu untuk

tiap element yang ditambahkan itu. Masing-masing ditambahkan satu sehingga meningkat menjadi 6 jumlahnya.

Kita bisa menyelesaikan masalah dengan membuang method *override* buat *addAll,* untuk sementara hasilnya akan benar, aplikasi jalan dengan baik, itu akan tergantung pada fungsi yang tepat, yang faktanya adalah method *addAll* yang ada pada *superclass-nya, HashSet.* Tapi masih ada masalah, tidak ada jaminan bahwa implementasi *HashSet* tidak akan berubah dari waktu ke waktu, bisa saja pada release berikutnya, fungsional dan beberapa fungsi dari method itu berubah.

Akan sedikit lebih baik untuk *override addAll method* untuk melakukan iterasi pada *collection* yang spesifik, panggil method *add* sekali saja untuk tiap element. Ini akan menjamin hasil yang benar, apakah ini method *HashSet addAll* atau tidak, yang mana method *addAll* tidak lagi dipanggil. Teknik ini, bagaimanapun tidak menyelesaikan masalah. Terdapat kekurangan pada cara jenis ini, yang sulit, waktu terlalu lama, rawan error, ini tidak selalu mungkin, sebagaimana semua method tidak bisa di-implementasikan tanpa akses ke *field* yang *private* yang tidak bisa di-akses dari *subclass.*

Issue yang menjadi issue pada sub-kelas adalah ketika super-kelas bisa mengganti method baru, method ini bisa saja menjadi cara baru untuk insert data ke collection, masalahnya adalah, ketika sub-kelas tidak meng-override method ini, dan pada release baru, super-kelas membuat method baru, akan terjadi kemungkinan penambahan data secara illegal, karena tidak sesuai dengan best practice yang baru, atau bisa juga method lama telah deprecated dan mesti menggunakan method yang baru.

Dua buah issue di-atas muncul dari override method. Kita mungkin berpikir bahwa lebih aman jika kita meng-extends sebuah kelas jika hanya membuat method yang baru dan menahan diri untuk tidak meng-override existing method. Meskipun cara ini terlihat lebih aman, tapi tetap saja terdapat bahaya yang mengintai, seandainya super-kelas mengganti dengan method

yang baru pada sub-kelasnya, dan kita sialnya memiliki method yang sama, hanya berbeda return type-nya, sub-kelas kita tidak akan jalan. Seandainya kita memiliki method dengan nama yang sama dan return type yang sama, maka kita sama saja dengan meng-override method di super-kelas tadi.

Ada cara untuk menghindari issue di-atas.

1. Buat sebuah private field pada kelas kita yang baru yang mereference pada kelas yang sudah ada. Namanya composition, karena kelas yang sudah ada menjadi bagian dari kelas yang baru.
2. Tiap-tiap method pada kelas yang baru memanggil method yang berkaitan pada instance kelas yang sudah ada/existing dan mengembalikan nilai. Namanya forwarding, dan method yang menggunakan itu pada kelas yang baru disebut forwarding method. Hasil akhirnya akan sekeras batu, tidak ada keterkaitan dengan implementasi dari existing kelas. Bahkan menambahkan method baru juga tidak akan memberikan pengaruh pada kelas yang baru.

perhatikan kode program di-bawah ini sebagai pengganti dari InstrumentedHashSet. Menggunakan composition dan forwarding. Catat bahwa, kelas di bagi menjadi 2, kelas itu sendiri (InstrumentedHashSet) dan penggunaan ulang dari forwarding class, yang memiliki semua forwarding method.

```
// Wrapper class - uses composition in place of inheritance
public class InstrumentedSet<E> extends ForwardingSet<E> {
        private int addCount = 0;
        public InstrumentedSet(Set<E> s) {
                super(s);
        }
        @Override
        public boolean add(E e) {
```

```
            addCount++;
            return super.add(e);
        }
        @Override
        public boolean addAll(Collection<? extends E> c) {
            addCount += c.size();
            return super.addAll(c);
        }
        public int getAddCount() {
            return addCount;
        }
}
// Reusable forwarding class
public class ForwardingSet<E> implements Set<E> {
        private final Set<E> s;
        public ForwardingSet(Set<E> s) {
            this.s = s;
        }
        public void clear() {
            s.clear();
        }
        public boolean contains(Object o) {
            return s.contains(o);
        }
        public boolean isEmpty() {
            return s.isEmpty();
        }
        public int size() {
```

```
        return s.size();
}
public Iterator<E> iterator() {
        return s.iterator();
}
public boolean add(E e) {
        return s.add(e);
}
public boolean remove(Object o) {
        return s.remove(o);
}
public boolean containsAll(Collection<?> c){
        return s.containsAll(c);
}
public boolean addAll(Collection<? extends E> c) {
        return s.addAll(c);
}
public boolean removeAll(Collection<?> c){
        return s.removeAll(c);
}
public boolean retainAll(Collection<?> c){
        return s.retainAll(c);
}
public Object[] toArray() {
        return s.toArray();
}
public <T> T[] toArray(T[] a) {
        return s.toArray(a);
```

```
    }
    @Override
    public boolean equals(Object o) {
        return s.equals(o);
    }
    @Override
    public int hashCode() {
        return s.hashCode();
    }
    @Override
    public String toString() {
        return s.toString();
    }
}
```

Design dari kelas InstrumentedSet dimungkinkan dari interface Set yang existing, yang membawa fungsionalitas dari kelas HashSet. Disamping robust, design ini sangat fleksibel. Kelas InstrumentedSet meng-implement interface Set dan memiliki konstruktor tunggal yang salah satu parameternya adalah Set. Esensinya, kelas mentransformasi satu set ke set lainnya, menambah fungsionalitasnya. Tidak seperti inheritance standar, yang bekerja hanya untuk sebuah concrete class dan menuntut sebuah konstruktor terpisah untuk setiap konstruktor pendukung di super-kelas, kelas wrapper bisa digunakan untuk meng-intrumentasikan setiap implementasi dari Set, maksudnya tiap kelas yang meng-implement atau turunan dari Set dan akan bekerja sama dengan konstruktor yang sudah ada sebelumnya.

```
Set<Date> s = new InstrumentedSet<Date>(new TreeSet<Date>(cmp));
Set<E> s2 = new InstrumentedSet<E>(new HashSet<E>(capacity));
```

Simpulannya, pewarisan adalah sangat powerful, tapi itu bermasalah karena itu melanggar enkapsulasi. Inheritance hanya tepat ketika hubungan sub-tipe asli ada antara subclass dan superclass. Bahkan kemudian, inheritance mungkin membawa pada kerapuhan jika sub-kelas berada pada package yang berbeda dengan super-kelas dan super-kelas tidak dirancang untuk melakukan pewarisan. Untuk menghindari kerapuhan ini, gunakan composition dan forwarding ketimbang inheritance, lebih khusus, adalah sebuah keputusan yang tepat, jika menggunakan wrapper kelas yang telah ada. Wrapper kelas tidak hanya lebih robust daripada sub-kelas, mereka juga lebih powerful.

Item 17. Design and document for inheritance or else prohibit it

Item di-atas sebagai peringatan tentang bahaya inheritance kalau tanpa rancangan dan dokumentasi.

1. kelas harus mendokumentasikan penggunaan yang dilakukannya terhadap method yang di-override. Kelas harus mendokumentasikan cara dan darimana saja proses pemanggilan method yang di override itu.
2. sebenarnya bukan hanya sekedar dokumentasi, tapi sebuah kelas mungkin saja memiliki keterkaitan dengan performa internal sebuah proses, sehingga kita harus memilih method protected dengan bijaksana.

   Contohnya, method :*protected void removeRange(int fromIndex, int toIndex*). Method akan melakukan removing pada data yang dimulai dari *fromIndex*sampe *toIndex.* Method ini akan dipakai oleh clear method untuk membersihkan data. Method ini mungkin tidak berpengaruh pada end user yang menggunakan list, tapi berpengaruh pada kelas yang menjadi sub-kelas dari kelas yang memilikinya. Dengan tidak adanya method ini, akan membuat proses clearing data menjadi lebih lambat.

terus bagaimanan cara agar kita mengetahui method*protected*mana yang harus kita expose ketika kita mendesign sebuah kelas untuk pewarisan. Sayangnya, tidak ada cara ajaib untuk melakukannya. Cara terbaik yang bisa dilakukan adalah, pikirkan dengan seksama, berikan tebakan terbaik kita dan kemudian buatlah sub-kelasnya. Kita harus meng-expose sesedikit mungkin *member protected,* karena masing-masing dari mereka merepresentasikan secara detail. Di sisi lain, kita tidak bisa meng-expose terlalu sedikit, sebagaimana sebuah kehilangan protected method bisa membuat proses pewarisan menjadi tidak berguna.

3. Satu-satunya cara untuk menguji apakah kelas dirancang untuk pewarisan adalah dengan membuat sub-kelasnya. Jika kita menghilangkan anggota dengan member protected yang krusial, cobalah untuk menuliskan sebuah sub-kelas yang akan membuat kelalaian berakibat menyakitkan yang sangat jelas. Sebaliknya, jika beberapa sub-kelas yang ditulis dan tidak ada yang menggunakan modifier protected, kita seharusnya membuat itu mungkin menjadi private. Pengalaman menunjukkan bahwa ketiga jenis sub-kelas di-atas biasanya cukup untuk menguji kelas yang bisa di-extends. Satu atau lebih dari tiga jenis sub-kelas itu harus dituliskan oleh seseoarang selain penulis super-kelasnya. Ketika kita merancang untuk pewarisan sebuah kelas yang cukup mungkin untuk digunakan secara luas.

4. Kita harus melakukan testing dengan membuat sub-kelas sebelum kita merelease code. Catat bahwa dokumentasi yang special sangat dibutuhkan untuk pewarisan, yang di-rancang untuk programmer yang membuat instance dari kelas kita dan memanggil method dengan instance itu. Sebagaimana tulisan ini, ada sedikit hal yang membedakan dokumentasi biasa dengan yang disiapkan untuk programmer dalam cara menggunakan tool atau komentar pada kode program yang dikhususkan untuk sub-classing.

5. Konstruktor tidak boleh memanggil method yang bisa di-override. Jika kita tidak mematuhi aturan ini, kegagalan pada aplikasi akan kita dapatkan. Kontruktor milik super-kelas akan running sebelum konstruktor sub-kelas, sehingga method yang di-override pada sub-kelas akan jalan sebelum konstruktor pada sub-kelas jalan. Jika method yang di-override bergantung pada proses inisialisasi pada konstruktor sub-kelas, maka method tidak akan running seperti seharusnya.

```
public class Super {
        // Broken - constructor invokes an overridable method
        public Super() {
                overrideMe();
        }
        public void overrideMe() {
        }
}

public final class Sub extends Super {
        private final Date date; // Blank final, set by constructor
        Sub() {
                date = new Date();
        }
        // Overriding method invoked by superclass constructor
        @Override
        public void overrideMe() {
                System.out.println(date);
        }
        public static void main(String[] args) {
                Sub sub = new Sub();
```

```
            sub.overrideMe();
        }
    }
```

5. Tidak satupun dari method*clone*, atau*readObject*bisa memanggil overridable method, langsung maupun tidak langsung.Untuk kasus readObject method, method yang di-override akan running sebelum sub-kelasnya di-deserialisasi. Pada kasus clone method, method yang di-override akan jalan sebelum method clone pada sub-kelas selesai untuk melakukan clone. Dan kondisi keduanya berpotensi untuk menimbulkan bug.
6. Merancang sebuah kelas untuk pewarisan menempatkan batasan yang substansial pada kelas. Sering sekali terjadi kesalahan ketika client mencoba untuk meng-extends kelas yang ada, yang menimbulkan bug/issue baru jika kita membiarkan mereka bisa mengakses berbagai method yang mutable, batasan di sini maksudnya adalah sebuah kelas bisa dibuat untuk memiliki field yang bersifat final, agar tidak bisa dimodifikasi oleh client nilainya, semata-mata untuk mencegah bug muncul pada saat maintenance.

7. Melarang membuat sub-kelas pada kelas yang tidak dirancang untuk dilakukan sub-kelas secara aman.

   Ada 2 cara untuk mencegah dari proses sub-classing.

   1. Cara termudah ada membuat kelas menjadi final.
   2. Cara alternative adalah membuat konstruktor menjadi provate atau package-private
   3. Lalu menambahkan public static factory sebagai pengganti konstruktor.

   Alternatif dua dan tiga menyediakan fleksibiltas untuk menggunakan sub-kelas hanya pada spesifik package tertentu saja.

Item 18. Prefer interface over to abstract class

Java menyediakan 2 jenis cara untuk mendefinisikan sebuah tipe atau kelas dengan banyak implementasi, yaitu dengan menggunakan Interface dan Abstract kelas. Perbedaan yang sangat jelas antara abstrak kelas dan interface adalah, abstrak kelas mengizinkan kita untuk hanya mengimplementasikan beberapa method miliknya saja, sedangkan interface tidak begitu. Satu lagi perbedaan penting adalah untuk meng-implementasikan sebuah tipe dari abstrak kelas, sebuah kelas tertentu harus menjadi subkelas dari abstrak kelas itu. Setiap kelas yang mendefinisikan semua method yang dibutuhkan atau diharuskan dan mematuhi aturan umumnya dibolehkan untuk meng-implement sebuah interface, tanpa memperhatikan dimana letak kelas itu dalam hirarki. Karena java hanya mengizinkan pewarisan tunggal, pembatasan ini pada kelas abstrak sangat membatasi penggunaannya sebagai jenis definition.

1. Kelas yang sudah exist bisa dengan mudah untuk menggunakan interface baru. Hanya dengan menambahkan klausa implement dan menambahkan semua method yang dibawa interface dengan override, kita sudah bisa menggunakan interface itu, berbeda dengan abstrak kelas, kita mesti memperhatikan hirarki dari kelas itu untuk melakukan extends, terlabih lagi kalau kita ingin meng-extends lebih dari satu abstrak kelas, kita mesti mengubah hirarki, karena java tidak mengizinkan extends lebih dari satu abstrak kelas.
2. Interface sangat ideal untuk mendefinisikan mixin. Mixin adalah sebuah tipe kelas yang bisa mengimplmentasikan fungsi tambahan pada fungsi utamanya,sebagai contoh interface Comparable yang mengizinkan kelas yang mendeklarasikannya untuk diurutkan atau dibandingkan dengan instance kelas sejenis. Sedangkan abstrak kelas tidak bisa melakukan itu, sebagaimana abstrak tidak bisa seketika meng-extends seperti penjelasan pada point nomer 1
3. Interface mengizinkan framework dibentuk tanpa hirarki. Hirarki sangat baik untuk meng-organize sesuatu, tapi tidak setiap hal bisa

dilakukan peng-hirarkian. Sebagai contoh kita memiliki Interface singer dan yang lainnya untuk songwriter.

```
public interface Singer {
        AudioClip sing(Song s);
}
public interface Songwriter {
        Song compose(boolean hit);
}
```

Pada kenyataanya, sangat mungkin penyanyi adalah juga seorang penulis lagu, karena kita menggunakan interface daripada abstrak kelas, sangat dimungkinkan untuk meng-implment kedua interface di-atas, atau kita juga bisa membuat interface baru yang meng-implement kedua interface di-atas dan menambahkan method baru sebagai representasi kombinasi dari Singer dan SongWriter.

```
public interface SingerSongwriter implements Singer, Songwriter {
        AudioClip strum();
        void actSensitive();
}
```

4. Interface mengaktifkan keamanan, fungsi pengembangan yang powerful dengan idiom wrapper class pada item 16. Jika menggunakan abstrak kelas, kita membuat programmer tidak punya pilihan lain selain menggunakan pewarisan bila ingin menambah fungsionalitas dengan cara mendefinisikan tipe tertentu. Menghasilkan kelas yang tidak powerful dan lebih rapuh daripada kelas wrapper. Sementara interface tidak membolehkan untuk memiliki implementasi method, menggunakan interface untuk mendefinisikan tipe tidak mencegah kita dari menyediakan implementasi untuk membantu programmer. Kita bisa mengkombinasikan kelebihan dari interface dan abstrak kelas dengan meyediakan implementasi kerangka kelas abstrak untuk ikut dengan setiap interface yang biasa saat kita eksport.

   Implementasi kerangka kelas abstrak adalah abstrack interface yang mana, interface adalah nama dari interface yang di-implementasi,

contohnya, Framework Collections menyediakan AbstractCollection, AbstractSet, AbstractList, and AbstractMap.

Dengan design yang tepat skeletal implementation bisa memberi programmer kemudahan untuk membuat implementasi mereka sendiri dari interface tertentu. Contohnya, disini ada sebuah static factory method yang memiliki fungsi List secara lengkap.

```
// Concrete implementation built atop skeletal implementation
static List<Integer> intArrayAsList(final int[] a) {
        if (a == null)
                throw new NullPointerException();
        return new AbstractList<Integer>() {
                public Integer get(int i) {
                        return a[i]; // Autoboxing (Item 5)
                }
                @Override
                public Integer set(int i, Integer val) {
                        int oldVal = a[i];
                        a[i] = val; // Auto-unboxing
                        return oldVal; // Autoboxing
                }
                public int size() {
                        return a.length;
                }
        };
}
```

Skeletal implementation adalah kelas abstrak yang diciptakan dengan meng-implementasi interface tertentu, tujuannya dan merupakan kelebihannya adalah, dia tidak memaksa kelas yang meng-extends dirinya untuk menggunakan method atau sejenisnya yang tidak sesuai atau membahayakan kelas itu sendiri, fungsinya atau cara membuat dan cara kerjanya sama dengan kelas abstrak pada umumnya, tapi kelas abstrak ini telah memperoleh definisi tambahan dari interface karena telah meng-implement interface tertentu. Contohnya seperti dibawah ini.

```
// Skeletal Implementation
```

```
public abstract class AbstractMapEntry<K,V> implements Map.Entry<K,V> {
        // Primitive operations
        public abstract K getKey();
        public abstract V getValue();

        // Entries in modifiable maps must override this method
        public V setValue(V value) {
                throw new UnsupportedOperationException();
        }

        // Implements the general contract of Map.Entry.equals
        @Override
        public boolean equals(Object o) {
                if (o == this)
                        return true;
                if (! (o instanceof Map.Entry))
                        return false;

                Map.Entry<?,?> arg = (Map.Entry) o;

                return equals(getKey(), arg.getKey()) && equals(getValue(),
        arg.getValue());
        }
        private static boolean equals(Object o1, Object o2) {
                return o1 == null ? o2 == null : o1.equals(o2);
        }
        // Implements the general contract of Map.Entry.hashCode
        @Override
         public int hashCode() {
                return hashCode(getKey()) ^ hashCode(getValue());
        }
        private static int hashCode(Object obj) {
                return obj == null ? 0 : obj.hashCode();
        }
}
```

Penggunaan skeletal implementation sebaiknya dilakukan jika kita ingin memiliki method yang tidak terlalu penting ketika akan melakukan eksport

sebuah kelas yang method itu didapatkan dari interface, tidak penting namun perlu. Kita sangat perlu untuk mempertimbangkan menggunakan ini.
Tidak selamanya menggunakan abstrak kelas itu merupakan kesalahan atau kerugian, ada sebuah kelebihan menggunakan abstrak kelas daripada interface, interface sangat sulit untuk dikembangkan dalam artian, sekali interface sudah dirilis, kita akan sangat sulit untuk melakukan modifikasi, penambahan method atau pengurangan method, karena ketika itu kita lakukan itu akan berakibat fatal pada semua kelas yang pernah meng-implement interface tersebut, ketika mereka melakukan compile kelas mereka pasti tidak jalan dan error, karena kelas itu tidak memiliki override terhadap method baru, atau masih meng-override method yang sudah tidak ada lagi pada interface tersebut. Jadi kita harus sangat yakin bahwa interface itu sudah benar dan tepat sebelum melakukan perilisan.
Simpulannya, masing-masing dari interface, abstrak kelas, atau skeletal implementation memiliki kelebihan dan kekurangan pada berbagai situasi, pilihlah salah satu dari mereka secara tepat, agar bisa memudahkan programmer atau kita untuk bekerja menyelesaikan task.

Item 19. Gunakan interface hanya untuk mendefinisikan type

Ketika class meng-implements sebuah interface, tujuannya adalah untuk membuat instance dari interface melalui kelas itu. Maksudnya adalah user atau client bisa menggunakan interface melalui kelas itu. Untuk tujuan lain sangat tidak dianjurkan, misalnya untuk mendefinisikan atribut yang berupa konstanta, itu terlalu sia-sia.

```
// Constant interface antipattern - do not use!
public interface PhysicalConstants {
    // Avogadro's number (1/mol)
    static final double AVOGADROS_NUMBER = 6.02214199e23;
    // Boltzmann constant (J/K)
    static final double BOLTZMANN_CONSTANT = 1.3806503e-23;
    // Mass of the electron (kg)
```

```
    static final double ELECTRON_MASS = 9.10938188e-31;
}
```

Contoh di-atas adalah contoh interface yang salah, ketika kita membuat interface yang akan di-implement berisikan hanya atribut saja. ***'The constant interface pattern is a poor use of interfaces'.***

Seandainya kita memang mutlak butuh untuk meng-eksport konstanta itu, maka kita bisa menggunakan tipe enum(Item 30) atau membuat sebuah kelas yang tidak bisa di-inisialisasi.

```
// Constant utility class
package com.effectivejava.science;
public class PhysicalConstants {
    private PhysicalConstants() { } // Prevents instantiation

    public static final double AVOGADROS_NUMBER = 6.02214199e23;
    public static final double BOLTZMANN_CONSTANT = 1.3806503e-23;
    public static final double ELECTRON_MASS = 9.10938188e-31;
}
```

Pada umumnya sebuah kelas konstanta menginginkan kelas yang secara langsung merepresentasikan namanya atau fungsinya,PhysicalConstants.AVOGADROS_NUMBER. Untuk merealisasikannya kita bisa menggunakan kode program seperti di bawah ini.

```
// Use of static import to avoid qualifying constants
import static com.effectivejava.science.PhysicalConstants.*;

public class Test {
    double atoms(double mols) {

        return AVOGADROS_NUMBER * mols;
    }
    ...
    // Many more uses of PhysicalConstants justify static import
}
```

Perhatikan saat kita meng-import library untuk kelas*PhysicalConstants*, setelah kata import kita menggunakan *modifier static*, sehinggan name dari field di kelas*PhysicalConstants*bisa dipanggil secara langsung menggunakan nama kelas dan nama fieldnya.

Simpulannya, interface seharusnya hanya digunakan untuk mendefinisikan tipe kelas interface tertentu. Mereka tidak seharusnya digunakan untuk meng-eksport konstanta.

Item 20. Prefer class hirarki daripada tagged class.

Tagged class di sini maksudnya adalah banyak terdapat kelas di dalam sebuah kelas, misalnya sebuah kelas bernama Bentuk, lalu di dalamnya kita bisa membuat bermacam-madam bentuk, seperti lingkaran dan kotak, jadi terdapat lebih dari satu instance yang bisa di buat dalam kelas itu, dan itu merupakan kelas yang buruk, kita lebih baik membuat semacam kelas hirarki, misalnya dengan membuat abstract bentuk, lalu lingkaran dan kotak meng-extends abstract class itu, sehingga terlihat semacam hirarki, dan lebih efektif kelasnya.

```
// Tagged class - vastly inferior to a class hierarchy!
class Figure {

        enum Shape { RECTANGLE, CIRCLE };

        // Tag field - the shape of this figure
        final Shape shape;

        // These fields are used only if shape is RECTANGLE
        double length;
        double width;

        // This field is used only if shape is CIRCLE
        double radius;

        // Constructor for circle
        Figure(double radius) {
                shape = Shape.CIRCLE;
                this.radius = radius;
        }

        // Constructor for rectangle
        Figure(double length, double width) {
```

```
            shape = Shape.RECTANGLE;
            this.length = length;
            this.width = width;
        }

        double area() {
            switch(shape) {
                case RECTANGLE:
                    return length * width;
                case CIRCLE:
                    return Math.PI * (radius * radius);
                default:
                    throw new AssertionError();
            }
        }
}
```

Jika kita perhatikan kelas di-atas, kelas itu terlihat normal, tapi sebenanrnya tidak normal, bagaimana sebuah kelas mempunyai interpretasi yang bermacam-macam, di dalam sebuah kelas Figure kita bisa membuat instance dari Figure yang berupa Circle dan Rectangle, belum lagi ada sebuah tipe enum untuk menterjemahkan jenis dari figure itu. Kelas seperti di atas sangat buruk dan berbahaya. Konstruktor yang dimiliki ada dua jenis, dengan argument yang dilempar berbeda-beda, bagaimana seandainya kita lupa atau salah melempar jenis argument, compiler tidak ada mekanisme untuk memberikan warning bahwa itu salah, dan ketika kita menjalankan program, maka akan memberikan banyak error. Belum lagi seandainya kita akan menambah atau mengembangkan jenis figure baru, tidak hanya dua, kelas itu akan di modifikasi lagi, menambah jenis tipe enum dan membuat konstruktor baru. Betapa sangat jelek design dari kelas itu.

Untungnya java menyediakan alternative untuk kelas seperti di-atas untuk diperbaiki yaitu dengan menggunakan kelas hirarki.

1. Kita membuat kelas abstrak untuk Figure. Kalau ada field yang tidak bergantung pada salah satu dari jenis tag, circle atau rectangle, dan bersifat umum, letakkan pada abstrak ini. Sama juga seandainya ada field yang dimiliki oleh keduanya, atau bersifat umum letakkan juga di dalam kelas abstrak ini.

```
// Class hierarchy replacement for a tagged class
abstract class Figure {
        abstract double area();
}
```

2. Selanjutnya buatlah kelas Circle dan Rectangle yang meng-extends abstrak kelas di-atas sebagai induknya, dan mereka menjadi anak dari kelas abstrak itu. Seperti inilah hirarki yang dimaksud, karena circle dan rectangle merupakan bagian dari figure, sehingga figure menjadi induknya.

```
class Circle extends Figure {
        final double radius;
        Circle(double radius) { this.radius = radius; }
        double area() { return Math.PI * (radius * radius); }
}

class Rectangle extends Figure {
        final double length;
        final double width;

        Rectangle(double length, double width) {
                this.length = length;
                this.width = width;
        }
        double area() { return length * width; }
}
```

Simpulannya,tagged class jarang sekali tepat penggunaannya. Jika kita ingin membuat sebuah tagged class, pikirkan tentang apakah tag di kelas itu bisa dibuang dan diganti dengan menggunakan kelas hirarki. Ketika kita menemukan kelas yang telah ada dengan sebuah atau banyak tag field, pertimbangkan untuk melakukan code refactoring menjadi kelas hirarki.

Item 21. Use function objects to represent strategies

Beberapa bahasa pemrograman mendukung fungsi pointer, delegates, lambda expression atau fasilitas sejenis yang mengizinkan program untuk menyimpan dan mentransfer kemampuan untuk memanggil fungsi tertentu.

Contohnya, fungsi *qsort* pada library standar bahasa C membutuhkan sebuah pointer untuk fungsi Comparator, yang *qsort* gunakan untuk membandingkan elemen agar terurut. Fungsi comparator membutuhkan 2 parameter, tiap-tiap mereka adalah pointer untuk elemen. Akan mengembalikan negative jika elemen pertama lebih kecil dari elemen kedua, nol jika elemen satu dan dua adalah sama, dan nilai positif jika elemen pertama lebih besar dari elemen kedua. Mekanisme pengurutan yang berbeda bisa diperoleh dengan melewatkan atau menggunakan parameter yang sama pada fungsi comparator yang berbeda. Ini namanya Strategy Pattern, fungsi Comparator merepresentasikan sebuah strategy untuk mengurutkan elemen.

Java tidak menyediakan fungsi pointer, tapi reference dari objek bisa digunakan untuk mendapatkan hasil yang sama. Pemanggilam sebuah method di dalam sebuah objek biasanya melakukan beberapa operasi tertentu pada objek itu. Bagaimanapun, mungkin saja untuk mendefinisikan sebuah objek yang methodnya melakukan operasi tertentu terhadap objek lainnya, dipasssing secara eksplisit ke method. Sebuah instance dari kelas yang notabene adalah objek yang digunakan oleh sebuah method tertentu, pada dasarnya adalah pointer ke objek itu melalui method. Beberapa instance diketahui sebagai *function objects.*

```
class StringLengthComparator {
        public int compare(String s1, String s2) {
                return s1.length() - s2.length();
        }
}
```

Kelas di-atas mem-public-kan sebuah method compare yang akan mengembalikan nilai negatif jika string pertama tidak lebih panjang dari string kedua, nol jika sama panjangnya, atau positif jika string pertama lebih panjang dari string kedua. Method ini adalah sebuah Comparator/pembanding yang membandingkan berdasarkan panjang

daripada berdasarkan lexicographic. Sebuah reference untuk objek StringLengthComparator ditempatkan sebagai sebuah fungsi pointer ke comparator itu, mengizinkan itu untuk dipanggil oleh String secara sepihak. Dengan kata lain, sebuah instance StringLengthComparator atau objek StringLengthComparator adalah sebuah concrete strategy untuk perbandingan string.

Sebagaimana sebuah kelas concrete strategy, kelas StringLengthComparator adalah stateless. Tidak memiliki field, karenanya sebuah objek kelas itu fungsinya sama. Jadi, mereka seharusnya adalah sebuah singleton untuk menjaga dari pembentukan objek yang tidak perlu.

```
class StringLengthComparator {
        private StringLengthComparator() { }

        public static final StringLengthComparator INSTANCE = new
StringLengthComparator();

        public int compare(String s1, String s2) {
                return s1.length() - s2.length();
        }
}
```

Untuk menggunakan sebuah objek StringLengthComparator pada sebuah method, kita memerlukan sebuah tipe yang tepat sebagai parameter. Mereka akan tidak bagus untuk digunakan karena client tidak akan bisa passing semua comparison strategy yang lainnya. Sebagai gantinya, kita perlu untuk medefinisikan sebuah interface Comparator dan memodifikasi StringLengthComparator jadi meng-implement interface Comparator. Dengan kata lain, kita perlu untuk mendefinisikan sebuah strategy interface sebelum sampai pada kelas concrete strategy.

```
// Strategy interface
public interface Comparator<T> {
        public int compare(T t1, T t2);
```

```
}
```

Interface comparator di-atas merupakan bagian dari java.util, tapi tidak ada yang aneh dengan itu. Kita juga bisa membuat sendiri. Interface comparator bersifat generic atau umum, sehingga bisa diterapkan pada objek apapun selain string. Method compare perlu dua parameter dengan tipe T (tipe formal parameter) ketimbang String. Kelas StringLengthComparator di-atas bisa dibuat untuk implement Comparator<String>.

```
class StringLengthComparator implements Comparator<String> {
        // class body is identical to the one shown above
}
```

Kelas concrete strategy sering dideklarasikan menggunakan kelas anonim.

```
Arrays.sort(stringArray, new Comparator<String>() {
        public int compare(String s1, String s2) {
                return s1.length() - s2.length();
        }
});
```

Jika anda perhatikan, tidak ada nama kelas yang spesifik untuk sorting array pada kode program di-atas, itulah maksud dari kelas anonim untuk comparator.

Tapi perlu di-ingat menggunakan sebuah kelas anonim untuk comparator akan membuat instance berkali-kali jika kita memanggil lebih dari sekali, akan dipanggil dan di-create berulang, pertimbangkan untuk menyiapkan function object dalam bentuk private static final field dan bisa digunakan berulang. Keuntungan lain kita bisa memberikan nama yang merepresentasikan function object itu.

Karena, interface strategy menyajikan sebuah tipe untuk semua instance dari concrete strategy-nya, sebuah kelas concrete strategy tidak perlu di-buat public untuk mem-public-kan sebuah concrete strategy. Karenanya, sebuah host class bisa mem-public-kan sebuah public static field (atau

method static factory) yang memiliki tipe interface strategy, dan kelas concrete strategy bisa menjadi kelas bersarang yang bersifat private di dalam host itu.

Sebagai contoh, sebuah member kelas yang static digunakan sebagai reference untuk sebuah kelas anonim untuk mengizinkan kelas concrete strategy untuk meng-implement interface kedua, Serializable.

```
// Exporting a concrete strategy
class Host {
        private static class StrLenCmp implements Comparator<String>,
Serializable {
                public int compare(String s1, String s2) {
                        return s1.length() - s2.length();
                }
        }
        // Returned comparator is serializable
        public static final Comparator<String> STRING_LENGTH_COMPARATOR =
new StrLenCmp();
        ... // Bulk of class omitted
}
```

Simpulannya, fungsi utama dari fungsi pointer adalah untuk implementasi strategy pattern. Untuk meng-implement ini dalam java, buatlah interface sebagai representasi dari strategy, dan sebuah kelas yang meng-implement interface untuk tiap concrete strategy. Ketika sebuah concrete strategy hanya digunakan sekali saja, kita bisa membuat dan meng-instansiasi sebagai sebuah kelas anonim. Ketika concrete strategy dirancang untuk digunakan secara berulang, mereka pada umumnya dibuat sebagai member kelas yang bersifat private static dan dibuat public, sebagai static final field yang berperan sebagai tipe interface strategy.

Item 22. Prefer static member class daripada non static

Sebuah kelas bersarang didefinisikan bersama dengan kelas lainnya. Sebuah kelas bersarang seharusnya keberadaannya untuk berperan terhadap kelas

induknya/kelas yang ditumpanginya. Jika sebuah kelas bersarang akan menjadi berguna dalam beberapa konteks lainnya, seharusnya dia menjadi kelas yang utama, bukan lagi kelas bersarang. Ada 4 jenis kelas bersararng.

1. Static member class
2. Non-static member class.
3. Anonymous class
4. Local class.

Jenis yang pertama dikenal dengan inner kelas. Materi kali ini akan menjelaskan kapan untuk menggunakan jenis nested kelas ini dan kenapa inner kelas adalah jenis yang paling simple.

Sebuah inner kelas adalah jenis kelas bersarang yang paling simple. Mereka sama dengan kelas pada umumnya yang dideklarasikan didalam kelas lainnya, dan memiliki akses kepada semua anggota kelas yeng ditumpanginya, bahkan untuk member yang bersifat private sekalipun. Sebuah inner kelas adalah static member dari kelas yang ditumpanginya dan memiliki sifat yang sama dengan member static pada umumnya. Jika mereka bersifat private, maka cara mengakses-nya hanya bisa melalui kelas yang menjadi wadahnya.

Salah satu kegunaan umum dari inner kelas adalah sebagai public helper class, berguna hanya sebagai penghubung dengan kelas di-luarnya atau kelas yang memilikinya. Sebagai contohnya, bayangkan sebuah tipe enum yang memiliki operasi calculator. Kelas enum Operation seharusnya public static member class dari kelas Calculator. Client dari Calculator dapat menggunakan Calculator untuk memakai fungsi dari enum Operation, Calculator.Operation.PLUS and Calculator.Operation.MINUS.

Secara sintaks, pembeda antara static dan non-static hanya modifier static pada tiap deklarasinya. Disamping kesamaan sintaks, dua jenis nested kelas ini sangatlah berbeda. Tiap anggota non-static dari kelas secara implisist berkaitan dengan instance dari kelas yang menjadi wadahnya. Dengan method instance dari anggota kelas yang bersifat non-static , kita bisa memanggil method pada instance kelas yang menjadi wadahnya atau memperoleh reference ke instance kelas yang menjadi wadahnya

menggunakan qualified this. Jika sebuah instance dari nested kelas bisa hidup dalam isolasi oleh instance kelas pengurungnya, kemudian nested kelas harus menjadi static member class. Tidak mungkin untuk menciptakan sebuah instance kelas dari non-static kelas tanpa instance dari kelas pengurungnya.

Hubungan antara instance dari kelas non-static dan instance kelas yang menjadi wadahnya dibangun ketika pembentuknya diciptakan, tidak dapat dimodifikasi setelahnya. Normalnya, hubungan terbangun otomatis dengan memanggil sebuah konstruktor kelas non-static dari dalam metode yang dipanggil oleh instance dari kelas yang menjadi wadahnya. Hal ini dimungkinkan, meskipun jarang, untuk membuat hubungan manual dengan menggunakan command *enclosingInstance.new MemberClass(args)*. Sebagaimana yang kita harapkan, hubungan memerlukan space dalam instance dari kelas non-static dan menambah waktu untuk membuat instance-nya. Salah satu penggunaan umum terhadap inner kelas yang bersifat non-static yang didefinisikan sebagai sebuah Adapter yang mengizinkan sebuah instance dari outer kelas untuk dilihat sebagai sebuah instance dari beberapa kelas yang tidak ada keterkaitan.

```
// Typical use of a nonstatic member class
public class MySet<E> extends AbstractSet<E> {
        // Bulk of the class omitted
        public Iterator<E> iterator() {
                return new MyIterator();
        }
        private class MyIterator implements Iterator<E>{.
        }
}
```

Jika kita mendeklarasikan sebuah member dari kelas yang tidak membutuhkan akses pada instance kelas yang menjadi wadahnya, selalu jadikan dia sebagai static pada saat dideklarasikan, jadikan dia static dari pada menjadikan sebagai non-static. Jika kita menghilangkan modifier ini, tiap instance akan memiliki sebuah reference asing ke instance kelas yang

menjadi wadahnya. Menyimpan reference ini membutuhkan waktu dan space yang lebih, dan bisa menghasilkan instance kelas menjadi tertahan ketika reference itu seharusnya telah siap untuk masuk ke garbage collector. Dan seharusnya kita pernah memerlukan untuk alokasi sebuah instance tanpa sebuah instance dari kelas yang mewadahinya, kita tidak akan bisa untuk melakukannya juga, sebuah instance dari inner kelas yang non-static dibutuhkan untuk memiliki instance dari kelas yang mewadahinya.

Sebuah penggunaan umum atas inner kelas yang bersifat private dan static adalah untuk merepresentasikan komponen dari representasi objek oleh kelas yang mewadahinya. Sebagai contoh, pikirkan tentang instance dari Map, yang menghubungkan kunci dengan nilainya. Banyak implementasi dari map memiliki internal objek Entry sendiri untuk tiap key-value di dalam Map. Ketika tiap entry dihubungkan dengan sebuah Map, method dalam sebuah Entry(getKey, getValue, dan setValue) tidak perlu melakukan akses ke Map. Oleh karena itu, akan sangat percuma untuk menggunakan inner kelas yang bersifat non-static untuk merepresentasikan Entries, sebuah inner kelas yang bersifat private dan static adalah cara terbaik. Jika kita secara tidak sengaja menghilangkan modifier statis pada deklarasi Entry, map akan tetap bekerja, tapi tiap-tiap entry akan memiliki reference tidak berguna ke Map, yang akan membuang-buang space dan waktu.

Hal ini sangatlah penting untuk memilih inner kelas yang benar antara static dan non-static jika inner kelas yang sedang dipertanyakan adalah sebuah member yang bersifat public atau protected dari kelas yang di-public-kan. Dalam kasus ini, inner kelas adalah sebuah API yang di-public-kan dan tidak bisa diganti dari non-static menjadi static di dalam perilisan tanpa melanggar compaabilitas secara binary.

Kelas Anonim.

Kelas anonim adalah tidak seperti yang lain-nya di dalam bahasa java. Sebagaimana yang kita harapkan, sebuah kelas anonim tidak memiliki nama. Dia bukan pula bagian dari atau anggota dari kelas yang mewadahinya. Daripada dideklarasikan dengan anggota yang lainnya, dia dideklarasikan

dan di-instansiasi secara simultan pada saat ingin digunakan.Kelas anonim dibolehkan pada setiap saat dalam code program asalkan ekspresin kode programnya boleh. Kelas anonim memiliki kelas yang mewadahinya jika dan hanya jika mereka muncul dalam konteks sebagai non-static. Tapi bahkan jika mereka muncul sebagai static sekalipun, mereka tidak bisa memiliki member static apapun.

Ada banyak batasan dalam penerapan kelas anonim. Kita tidak bisa meng-instansiasi mereka kecuali pada saat mereka digunakan. Kita tidak bisa melakukan pengecekan ataupun test apapun yang diperlukan untuk mengetahui nama dari kelas itu. Kita tidak bisa mendeklarasikan sebuah kelas anonim untuk meng-implement multiple interface, atau untuk meng-extends sebuah kelas dan meng-implement sebuah interface pada saat yang bersamaan. Client dari kelas anonim tidak bisa memanggil member apapun kecuali mereka menerima warisan dari induknya. Karena kelas anonim muncul di tengah-tengah kode program, maksudnya ketika kita ingin menggunakannya sesaat setelah proses pembentukan kelas selesai, mereka harus tetap singkat, sekitar 10 baris atau kurang atau pembacaan kode program akan kesulitan.

1. Salah satu penggunaan yang umum untuk kelas anonim adalah membuat function objects. Sebagai contohnya, sebuah method sort, berguna untuk men-sorting array dari string berdasarkan panjang string itu menggunakan instance anonim Comparator.
2. Fungsi lainnya dari kelas anonim adalah untuk membuat process objects, misalnya instance Runnable, Thread, atau TimerTask.
3. Fungsi ketiga adalah dalam penggunaan static factory method.see the intArrayAsList method in Item 18

Kelas Local.

Kelas local setidaknya digunakan secara berkala oleh empat jenis kelas bersarang. Sebuah kelas local bisa dideklarasikan dimananpun sebuah local variable bisa dideklarasikan dan mematuhi aturan yang sama. Kelas local memiliki atribut yang umum dengan tiap-tiap dari mereka merupakan

bentuk dari kelas bersarang. Seperti kelas member, mereka memiliki nama dan bisa digunakan berulang. Seperti kelas anonim, mereka memiliki instance kelas yang mewadahinya jika mereka dideklarasikan dalam bentuk non-static, dan mereka tidak bisa memiliki member yang static. Dan seperti kelas anonim. Mereka seharusnya bisa tetap singkat sehingga tidak membahayakan kode program saat dibaca.

Sebagai pengulangan, ada 4 jenis kelas bersarang, dan tiap-tiap mereka memiliki tempatnya masing-masing. Jika sebuah kelas bersarang perlu untuk terlihat diluar dari method tunggal atau terlalu panjang untuk cocok dengan baik di dalam sebuah method, guakan sebuah member dari kelas. Jika tiap-tiap instance dari anggota kelas perlu sebuah reference ke kelas yang mewadahinya, jadikan mereka non-static, sebaliknya, jadikan static. Anggap kelas digunakan dari dalam method. Jika kita ingin untuk membuat instance dari hanya satu lokasi dan ada tipe yang eksis yang mencerminkan kelas itu, buatlah menjadi kelas anonim, sebaliknya buatlah menjadi kelas local.

BAB 5. Generic

Pada release java 5, generic telah ditambahkan ke java. Sebelum generic, kita harus melakukan casting pada setiap object yang dibaca dari collection. Jika seseorang tidak sengaja meng-insert sebuah objek dengan tipe yang salah, proses casting akan gagal saat runtime. Dengan generic, kita member tahu compiler apa tipe dari object yang dibolehkan untuk masing-masing collection. Compiler meng-insert dan casting otomatis dan member tahu kita saat waktu compile jika kita mencoba untuk insert objek dengan tipe yang salah. Hasilnya, program menjadi lebih aman dan clear, tapi kentungan ini hadir bersama dengan keruwetannya. Chapter ini akan membahas, bagaimana kita bisa untuk memaksimalkan keuntungan dan meminimalkan keruwetannya.

Item 23. Jangan gunakan raw type untuk code yang baru.

Sebenarnya maksudnya dari ini adalah gunakan fasitas generic yang sudah disediakan oleh Java, spesifiknya untuk collection data, seperti List,Set,Map dkk. Contohnya, List Nilai, dengan List<String> Nilai, list yang pertama, kita harus casting datanya ke String terlebih dahulu, untuk bisa dibaca, sehingga tidak efektif, sedangkan yang kedua, kita bisa langsung pakai nilainya, karena udah di declare yang bisa masuk ke sana hanyalah nilai String saja. Itu membuat kerjaan menjadi lebih efektif, juga mengurangi kemungkinan error, seandainya terjadi ketidaksesuaian tipe data ketika melakukan casting atau proses lainnya. Sebaiknya gunakan bukan raw data untuk data yang belum kelas tipenya, bisa di ganti dengan Set<?>, sehingga kalo kita melempar parameter, pakailah sejenis itu, Jangan Set yang umum seperti biasa.

```
// Now a raw collection type - don't do this!
/**
* My stamp collection. Contains only Stamp instances.
*/
private final Collection stamps = ... ;
```

Kode program di-atas tidak terlihat salah, hanya saja terdapat jebakan yang sangat berbahaya, seandainya kita menginsert objek dengan tipe koin ke dalam stamps, saat compile tidak akan terjadi apapun, kita tidak memperoleh warning apapun. Saat runtime pun ketika kita insert objek koin tidak akan menjadikan kita mendapatkan error atau exception.

```
// Erroneous insertion of coin into stamp collection
stamps.add(new Coin( ... ));
```

Masalah mulai muncul saat kita akan retrieve nilainya.

```
// Now a raw iterator type - don't do this!
for (Iterator i = stamps.iterator(); i.hasNext(); ) {
Stamp s = (Stamp) i.next(); // Throws ClassCastException
... // Do something with the stamp
}
```

Saat kita retrieve nilai, kita akan memperoleh error, karena kita casting objek koin ke stamp, yang mana keduanya tidak compatible, munculnya ClassCastException. Kita dipaksa mencari letak errornya dimana, compiler tidak bisa membantu untuk mencari dimana letak salah meng-insert nilai ke stamp collection, dan kita mendapati pesan : Contains only Stamp instances.

Dengan generic kita menghindarkan kasus itu, kita akan memberitahukan pada compiler tentang apa yang benar dan salah saat akan insert objek ke collection.

```
// Parameterized collection type - typesafe
private final Collection<Stamp>stamps = ... ;
```

Saat kita salah insert, meng-insert koin yang seharusnya stamps, maka compiler akan memberitahukan kita bahw kita salah insert.

```
Test.java:9: add(Stamp) in Collection<Stamp> cannot be applied
to (Coin)
stamps.add(new Coin());
```

Proses retrieve pun akan berlangsung lebih aman, tanpa embel-embel casting.

```
// for-each loop over a parameterized collection - typesafe
for (Stamp s : stamps) { // No cast
... // Do something with the stamp
}
```

Atau

```
// for loop with parameterized iterator declaration - typesafe
for (Iterator<Stamp>i = stamps.iterator(); i.hasNext(); ) {
Stamp s = i.next(); // No cast necessary
... // Do something with the stamp
}
```

Ada sebuah update terbaru lagi yang disediakan oleh generic, ketika kita ingin memiliki sebuah koleksi tapi kita tidak tahu persis tipe apakah yang

akan kita operasikan, maka tetap saja jangan gunakan raw type (List, Set, Map), kita bisa gunakan List<E>, Set<E>, Map<E> yang juga adalah List<?>, Set<?>, Map<?>. Mereka ini disebut dengan ***unbounded wildcard types***.

```
// Unbounded wildcard type - typesafe and flexible
static int numElementsInCommon(Set<?>s1, Set<?>s2) {
        int result = 0;
        for (Object o1 : s1)
                if (s2.contains(o1))
                        result++;
        return result;
}
```

Apa perbedaan Set<?> dan raw type Set ? wildcard type aman sedangkan raw type tidak. Kita bisa meletakkan elemen apapun ke dalam collection dengan raw type, dengan mudahnya membuat tipe collections menjadi inkonsisten, kita tidak bisa meletakkan elemen apapun ke dalam Collection<?> selain null.

```
WildCard.java:13: cannot find symbol
symbol : method add(String)
location: interface Collection<capture#825 of ?>
c.add("verboten");
```

kita meletakkan String sehingga kita memperoleh error saat di-compile.

Kita harus menggunakan raw type dalam kelas literal, karena parameterized tidak di-ijinkan. Dengan kata lain, List.class, String[].class dan int.class adalah legal semuanya, tapi List<String>.class dan List<?> tidak.
Aturan kedua yang dikecualikan dari aturan ini berfokus pada operator instanceof. Karena informasi tentang generic type akan hilang atau dihapus saat runtime, hal ini illegal untuk menggunakan instanceof pada parameterized type dari pada unbounded wildcard type. Penggunaan unbounded wildcard type pada tempat raw type tidak memberikan dampak

terhadap behavior dari operator instanceof. Dalam kasus ini, kurung lancip dan tanda tanya, hanya membuat ribet. Inilah cara yang dianjurkan untuk menggunakan operator instanceof dengan tipe generic.

```
// Legitimate use of raw type - instanceof operator
if (o instanceof Set) { // Raw type
        Set<?>m = (Set<?>) o; // Wildcard type
}
```

Catat, bahwa sekali kita telah memastikan bahwa o adalah sebuah Set, kita harus casting itu ke tipe wildcard Set<?>, bukan tipe raw Set. Ini adalah casting checking, sehingga tidak akan menyebabkan sebuah warning dari compiler.

Term	Example	Item
Parameterized type	`List<String>`	Item 23
Actual type parameter	`String`	Item 23
Generic type	`List<E>`	Items 23, 26
Formal type parameter	`E`	Item 23
Unbounded wildcard type	`List<?>`	Item 23
Raw type	`List`	Item 23
Bounded type parameter	`<E extends Number>`	Item 26
Recursive type bound	`<T extends Comparable<T>>`	Item 27
Bounded wildcard type	`List<? extends Number>`	Item 28
Generic method	`static <E> List<E> asList(E[] a)`	Item 27
Type token	`String.class`	Item 29

Item 24. Eliminasi Unchecked Warning.

Ketika kamu baru pemula dalam Generic, Jangan pernah berpikir untuk bisa compile code dengan bersih tanpa warning untuk penerapan Generic, baik dari segi deklarasi variabelnya, maupun dari saat akan casting data. Remove

sebanyak apapun unchecked warning yang kamu dapat, karena sering kali juga, error terjadi ketika runtime, yaitu ClassCastException.

```
Set<Lark> exaltation = new HashSet();
```

Compiler akan memberitahu bahwa kode di-atas salah.

```
Venery.java:4: warning: [unchecked] unchecked conversion
found : HashSet, required: Set<Lark>
Set<Lark> exaltation = new HashSet();
```

Lalu diperbaiki menjadi.

```
Set<Lark> exaltation = new HashSet<Lark>();
```

Beberapa warning akan sulit untuk diperbaiki, item ini penuh dengan contoh-contoh itu. Buang semua unchecked warning sebanyak yang kita bisa. Jika kita bisa melakukannya, kita setidaknya telah menjamin bahwa program kita aman, dan itu penting dan mencegah untuk tidak mendapatkan ClassCastException saat runtime.

Jika kita tidak bisa mengeliminasi sebuah warning, dan kita bisa membuktikan bahwa kode program yang ditulis sebagai warning adalah aman, kemudian tekankan keterangan pada warning dengan sebuah annotation **@SuppressWarnings("unchecked").** Jika kita menekankan warning tanpa pertama-tama membuktikan bahwa kode program itu aman, kita hanya sedang memberikan diri kita sebuah kesalahan dalam merasakan kemungkinan error. Kode program mungkin bisa di compile dengan aman tanpa warning apapun, tapi mereka masih tetap bisa mendapatkan sebuah ClassCastException saat runtime. Jika, bagaimanapun, kita mengabaikan unchecked warning yang kita tahu aman, kita tidak akan peduli ketika sebuah warning merepresentasikan problem yang sebenarnya.

```
public <T> T[] toArray(T[] a) {
        if (a.length < size)
                return (T[]) Arrays.copyOf(elements, size, a.getClass());
```

```
        System.arraycopy(elements, 0, a, 0, size);
        if (a.length > size)
                a[size] = null;
        return a;
}
```

Jika di-compile akan memberikan warning seperti di bawah ini.

```
ArrayList.java:305: warning: [unchecked] unchecked castfound : Object[], required: T[]
return (T[]) Arrays.copyOf(elements, size, a.getClass());
```

Adalah tindakan illegal untuk memberikan annotation suppress warning pada return value.

```
// Adding local variable to reduce scope of @SuppressWarnings
public <T> T[] toArray(T[] a) {
        if (a.length < size) {
                // This cast is correct because the array we're creating
                // is of the same type as the one passed in, which is T[].
                @SuppressWarnings("unchecked")
                 T[] result = (T[]) Arrays.copyOf(elements, size, a.getClass());
                return result;
        }
        System.arraycopy(elements, 0, a, 0, size);
        if (a.length > size)
                a[size] = null;
        return a;
}
```

Setiap kali kita memberikan annotation suppress warning, tambahkan komentar bahwa kode program itu aman, itu untuk memudahkan orang mengerti tentang kode program kita dan mencegah mereka melakukan modifikasi pada kode program yang membuat nantinya kode program itu menjadi tidak lagi aman.

Item 25. Prefer List to Arrays.

Lebih baik menggunakan list atau object collection lain yang bisa diterapkan dengan menggunakan konsep Generic, sehingga lebih memudahkan dalam proses instance sebuah variable, dan proses manipulasI nilainya, ketika kita melakukan casting data atau yang lainnya, sedangkan Array tidak berlaku untuk menggunakan Generic, dan dalam proses manipulasi data nilainya lebih besar kemungkinan untuk error atau exception ketika casting class atau objek dari nilai tertentu.

This code fragment is legal:

// Fails at runtime!

Object[] objectArray = new Long[1];

objectArray[0] = "I don't fit in"; // Throws ArrayStoreException

but this one is not:

// Won't compile!

List<Object> ol = new ArrayList<Long>(); // Incompatible types

ol.add("I don't fit in");

1. Dengan menggunakan array kita menemukan error atau exception saat runtime, sedangkan dengan list kita menemukan error saat compile time.

2. Array tahu dan memaksa tipe dari elementnya saat runtime. Jika kita mencoba untuk menyimpan String pada array yang bertipe Long, kita akan memperoleh ArrayStoreException Generic. Jelasnya di-implementasikan dengan menghapus. Maksudnya bahwa mereka memaksa type dari constraint mereka saat compile time dan menghapus tipe elemen mereka saat runtime. Menghapus adalah mengizinkan tipe generic untuk beroperasi secara bebas dengan kode yang legal dan benar yang tidak menggunakan generics. Karena fundamental mereka berbeda, array dan generic tidak bisa bercampur

dengan baik. Contohnya, adalah illegal untuk membuat array dengan tipe generic atau tipe parameter. Tak ada satupun dari ekspresi pembuatan array yang benar, new List<E>[], new List<String>[], new E[]. Semuanya menyebabkan error saat compile time. Kenapa illegal untuk membuat generic array ? karena itu tidaklah aman. Jika mereka dilegalkan, proses casting oleh compiler dalam program dinyatakan benar bisa gagal di saat runtime dengan ClassCastException. Ini akan melanggar jaminan dasar dari generic.
Sebagai contoh concete-nya.

```
// Why generic array creation is illegal - won't compile!
List<String>[] stringLists = new List<String>[1];       // (1)
List<Integer> intList = Arrays.asList(42);                  // (2)
Object[] objects = stringLists;                             // (3)
objects[0] = intList;                                       // (4)
String s = stringLists[0].get(0);                           // (5)
```

Ketika kita memperoleh error saat pembuatan array, maka cara terbaik adalah sering menggunakan tipe collection List<E> sebagai pengganti E[]. Kita mungkin mengorbankan peforma dan keringkasan, tapi sebagai gantinya kita mendapatkan safety and interoperability

```
// Reduction without generics, and with concurrency flaw!
static Object reduce(List list, Function f, Object initVal) {
        synchronized(list) {
                Object result = initVal;
                for (Object o : list)
                        result = f.apply(result, o);
                return result;
        }
}
interface Function {
        Object apply(Object arg1, Object arg2);
}
```

Jangan panggil method alien di dalam wilayah sinkronisasi. Sehingga, kita memodifikasi method reduce untuk meng-copy content dari list ketika

menahan kunci, yang mana mengizinkan kita untuk melakukan reduction dari hasil kopian.

```
// Reduction without generics or concurrency flaw
static Object reduce(List list, Function f, Object initVal) {
        Object[] snapshot = list.toArray(); // Locks list internally
        Object result = initVal;
        for (Object o : list)
                result = f.apply(result, o);
        return result;
}
```

Jika ingin mencoba melakukan ini dengan metode generics, kita akan memperoleh masalah saat sorting yang kita diskusikan sebelumnya.

```
interface Function<T> {
        T apply(T arg1, T arg2);
}

// Naive generic version of reduction - won't compile!
static <E> E reduce(List<E> list, Function<E> f, E initVal) {
        E[] snapshot = list.toArray(); // Locks list
        E result = initVal;
        for (E e : snapshot)
                result = f.apply(result, e);
        return result;
}
```

If you try to compile this method, you'll get the following error:

```
Reduce.java:12: incompatible types found : Object[], required: E[]
E[] snapshot = list.toArray(); // Locks list
```

No big deal, you say, I'll cast the Object array to an E array:

```
E[] snapshot = (E[]) list.toArray();
```

That gets rid of the error, but now you get a warning:

```
Reduce.java:12: warning: [unchecked] unchecked cast found : Object[], required:
E[]
E[] snapshot = (E[]) list.toArray(); // Locks list
```

Compiler sedang memberitahukan kita bahwa dia tidak bisa cek keamanan proses casting saat runtime karena dia tidak tahu apa tipe dari E saat runtime, ingat, informasi tentang tipe dari elemen dihapus dari generic saat runtime. Akankah program jalan ? ya, tapi itu tidak aman. Dengan sedikit modifikasi, kita bisa mendapatkan sebuah exception, ClassCastException pada sebuah line yang tidak memiliki eksplisit casting. Waktu compile, tipe dari snapshot adalah E[] yang mana bisa menjadi String[], Integer{} atau tipe lainnya. Tipe runtime adalah object[] dan itu berbahaya.

```
// List-based generic reduction
static <E> E reduce(List<E> list, Function<E> f, E initVal) {
        List<E> snapshot;
        synchronized(list) {
                snapshot = new ArrayList<E>(list);
        }
        E result = initVal;
        for (E e : snapshot)
                result = f.apply(result, e);
        return result;
}
```

Item 26. Favor Generic Type.

Usahakan untuk menggunakan tipe data yang bersifat generic, karena itu lebih aman dan lebih mudah, karena itu akan menghindari operasi casting nilai di sisi client yang mengurangi resiko ClassCastException, dikarenakan proses casting value yang bermasalah. Pastikan juga tipe data generic yang baru tidak menganggu kerja dari tipe data yang lama, sehingga bisa jalan tanpa merusak tipe data yang udah existing.

Sebenarnya tidak terlalu sulit untuk memberi parameter pada deklarasi collection kita dan membuat menggunakan tipe generic dan method yang disediakan oleh JDK. Membuat generic type sendiri sedikit lebih sulit, tapi masih setimpal dengan usaha utuk belajar bagaimana caranya.

```
// Object-based collection - a prime candidate for generics
public class Stack {
        private Object[] elements;
        private int size = 0;
        private static final int DEFAULT_INITIAL_CAPACITY = 16;
        public Stack() {
                elements = new Object[DEFAULT_INITIAL_CAPACITY];
        }
        public void push(Object e) {
                ensureCapacity();
                elements[size++] = e;
        }
        public Object pop() {
                if (size == 0)
                        throw new EmptyStackException();
                Object result = elements[--size];
                elements[size] = null; // Eliminate obsolete reference
                return result;
        }
        public boolean isEmpty() {
                return size == 0;
        }
        private void ensureCapacity() {
                if (elements.length == size)
                        elements = Arrays.copyOf(elements, 2 * size + 1);
        }
}
```

The next step is to replace all the uses of the type Object with the appropriate type parameter, and then try to compile the resulting program:

```
// Initial attempt to generify Stack = won't compile!
public class Stack<E>{
        private E[] elements;
        private int size = 0;
        private static final int DEFAULT_INITIAL_CAPACITY = 16;
        public Stack() {
                elements = new E[DEFAULT_INITIAL_CAPACITY];
```

```
    }
    public void push(E e) {
        ensureCapacity();
        elements[size++] = e;
    }
    public E pop() {
        if (size==0)
            throw new EmptyStackException();
        E result = elements[--size];
        elements[size] = null; // Eliminate obsolete reference
        return result;
    }
    ... // no changes in isEmpty or ensureCapacity
}
```

You'll generally get at least one error or warning, and this class is no exception. Luckily, this class generates only one error:

```
Stack.java:8: generic array creation
elements = new E[DEFAULT_INITIAL_CAPACITY];
```

1. Salah satu solusinya adalah membuat sebuah array object dan casting menjadi tipe generic.

```
Stack.java:8: warning: [unchecked] unchecked cast found : Object[], required: E[]
    elements = (E[]) new Object[DEFAULT_INITIAL_CAPACITY];
```

Tapi tetap saja mendapatkan warning, compiler tidak percaya dan yakin bahwa kode program kita ini aman. Tapi kita bisa, kita harus meyakinkan diri kita bahwa unchecked cast tidak akan membahayakan keamanan dari program kita. Array disimpan di dalam sebuah private field dan tidak pernah dikembalikan ke client atau dilewatkan ke method manapun. Element hanya disimpan dalam array yang mana mereka dilewatkan ke push method, yang bertipe E sehingga unchecked cast tidak akan membahayakan.

Sekali kita membuktikan bahwa unchecked cast aman, annotation suppress warning mungkin untuk digunakan. Dalam kasus ini, konstruktor hanya memiliki unchecked array creation, sehingga itu tepat untuk melakukan suppress warning pada keseluruhan konstruktor. Dengan penambahan sebuah annotation untuk melakukan itu, Stack akan di-compile aman, dan kita menggunakan-nya tanpa eksplisit casting atau ketakutan akan ClassCastException.

```
// The elements array will contain only E instances from push(E).
// This is sufficient to ensure type safety, but the runtime
// type of the array won't be E[]; it will always be Object[]!
@SuppressWarnings("unchecked")
public Stack() {
        elements = (E[]) new Object[DEFAULT_INITIAL_CAPACITY];
}
```

2. Cara untuk menghilangkan atau membuang proses pembentukan objek array generic yang error pada Stack adalah dengan mengganti tipe dari field elements dari E[] ke Object[], jika kita memilih cara ini, jika akan mendapatkan error yang berbeda.

```
Stack.java:19: incompatible types found : Object, required: E
E result = elements[--size];
```

Kita bisa saja membuang error dan mendapatkan warning dengan casting data elemen yang diambils dari array semula tipe object ganti ke E.

```
Stack.java:19: warning: [unchecked] unchecked cast found : Object, required: E
E result = (E) elements[--size];
```

Karena E adalah tipe non-reifiable, tidak ada cara compiler bisa mengetahui casting saat runtime. Sekali lagi, kita bisa dengan mudah membuktikan pada diri kita bahwa unchecked cast itu aman. Sehingga, adalah tepat menggunakan suppress warning. Pada point saran dari item 24, kita melakukan suppress warning hanya untuk proses assignment yang membawa proses unchecked cast, bukan pada keseluruhan method pop.

```
// Appropriate suppression of unchecked warning
public E pop() {
        if (size==0)
                throw new EmptyStackException();
        // push requires elements to be of type E, so cast is correct
        @SuppressWarnings("unchecked") E result = (E) elements[--size];
        elements[size] = null; // Eliminate obsolete reference
        return result;
}
```

Apapun cara yang kita pilih dari dua pilihan ini untuk mengatasi error yang terjadi saat instansiasi generic array adalah hanya masalah selera. Kesemuanya sama saja, lebih beresiko untuk melakukan suppress warning untuk sebuah unchecked cast pada sebuah array ketimbang pada scalar type, yang mana akan member solusi kedua. Tapi pada sebuah generic class yang lebih nyata daripada Stack, kita mungkin akan membaca dari array pada banyak tempat di kode program, sehingga memilih solusi kedua akan membutuhkan banyak proses casting ke E daripada sebuah casting tunggal ke E[], itulah kenapa solusi pertama lebih umum digunakan ketimbang solusi kedua.

Program berikut ini menjabarkan penggunaan kelas generic Stack. Program mencetak urutan terbalik dan membuat menjadi huruf besar semua. Tidak ada explicit casting yang layak untuk memanggil method konversi string ke huruf besar pada saat melakukan pop elemen dari Stack, dan otomatis melakukan casting untuk menjamin keberhasilan program.

```
// Little program to exercise our generic Stack
public static void main(String[] args) {
        Stack<String> stack = new Stack<String>();
        for (String arg : args)
                stack.push(arg);
        while (!stack.isEmpty())
```

```
            System.out.println(stack.pop().toUpperCase());
        }
```

Contoh sebelumnya mungkin terlihat kontradiksi dengan item 25, yang mendorong penggunaan list sebagai pilihan untuk array. Hal ini tidak selalu mungkin untuk menggunakan list dalam tipe generic kita. Java tidak mendukung native list, sehingga beberapa tipe generic, ArrayList, harus di-implementasi di-atas array. Tipe generic lain, misalnya sebagai HashMap, di-implementasikan di-atas array untuk mendukung peforma.

Mayoritas tipe generic adalah seperti contoh Stack kita bahwa parameter jenis mereka tidak memiliki batasan. Kita bisa membuat Stack<Object>, Stack<int[]>, Stack<List<String>>, atau sebuah stack dengan jenis reference lainnya. Catat bahwa kita tidak bisa membuat Stack dari tipe data primitive, mencoba membuat sebuah Stack<int>, Stack<double> akan membuat error saat compile. Ini adalah batasan mendasar dari penerapan java generic. Kita bisa mengganti tipe itu dengan tipe kelasnya, misalnya Stack<Integer> atau Stack<Double>.

Ada beberapa tipe generic yang melonggarkan batasan atas tipe dari nilai yang menjadi paramaternya. Contohnya, java.util.concurrent,DelayQueue, yang dideklarasikan seperti dibawah ini.

```
class DelayQueue<E extends Delayed>implements BlockingQueue<E>;
```

List dengan tipe parameter (E extends Delayed) menuntut tipe actual dari parameter E merupakan sub-type dari java.util.concurrent.Delayed. Ini mengizinkan implementasi terhadap DelayQueue dan client-nya untuk mengambil keuntungan atas method Delayed pada element dari DelayedQueue, tanpa perlu melakukan ekplisit casting atau resiko atas ClassCastException. Tipe dari parameter E diketahui adalah sebagai bounded type parameter. Catat, mengenai hubungan sub-type yang didefinisikan, sehingga untuk setiap tipe adalah sebuah sub-type dari dirinya. Sehingga normal dan benar untuk membuat DelayQueue<Delayed>.

Simpulannya, tipe generic aman dan lebih mudah untuk digunakan daripada tipe yang membutuhkan casting pada kode program di client. Ketika kita merancang tipe baru, pastikan bahwa mereka bisa digunakan tanpa casting apapun. Ini akan sering berguna dalam pembuatan tipe data generic. Ini akan membuat penggunaan lebih mudah untuk pengguna baru dari tipe-tipe itu tanpa merusak pengguna yang sudah eksis (item 23).

Item 27. Favor Generic Method.

Pembuatan generic method akan sama dengan pembuatan generic type.

```
// Uses raw types - unacceptable! (Item 23)
public static Set union(Set s1, Set s2) {
        Set result = new HashSet(s1);
        result.addAll(s2);
        return result;
}
```

Kita akan memperoleh dua buah warning saat compile kode program itu.

```
This method compiles, but with two warnings:
Union.java:5: warning: [unchecked] unchecked call to
HashSet(Collection<? extends E>) as a member of raw type HashSet
Set result = new HashSet(s1);
^
Union.java:6: warning: [unchecked] unchecked call to
addAll(Collection<? extends E>) as a member of raw type Set
result.addAll(s2);
```

Selain tipe data generic, kita juga bisa membuat generic method, sehingga tidak udah pusing untuk casting nilainya, misalnya contoh kodenya dibawah. Warning pada kode sebelumnya bisa di-atasi dengan.

```
// Generic method
public static <E>Set<E>union(Set<E>s1, Set<E>s2) {
        Set<E>result = new HashSet<E>(s1);
        result.addAll(s2);
        return result;
```

```
}
```

dengan adanya parameter E, yang mencerminkan generic, yang maksudnya Set bisa bertipe apa aja. Sehingga, ketika melempar nilai Set menjadi parameternya, maka method akan secara otomatis mengenali tipenya, dan mengembalikan return value, dengan tipe yang sama. Model ini dinamakan type inference, yang secara otomatis mengenali dan mengembalikan nilai yang diinginkan.

Contoh berikut ini, yang merupakan -generic static factory method-, yaitu.

```
// Generic static factory method
public static <K,V> HashMap<K,V> newHashMap() {
        return new HashMap<K,V>();
}

// Parameterized type instance creation with static factory
Map<String, List<String>> anagrams = newHashMap();
```

method ini akan mengembalikan nilai objek sesuai dengan yang diinginkan, dan kita tidak usah pula ketika akan menginstance objek ini, dengan statement new HashMap<tipe1, tipe2>, kita hanya perlu panggil static factory method newHashMap(), dan akan secara otomatis mengembalikan nilai objek dengan tipe yang sama.

Pattern yang mirip dengan static factory method adalah generic singleton factory. Pada umumnya, kita akan perlu untuk membuat objek yang immutable tapi berlaku untuk semua tipe yang berbeda-beda. Karena generic adalah implementasi dengan menghapus (item 25), kita bisa menggunakan objek tunggal untuk semua tipe parameter yang dibutuhkan, tapi kita perlu untuk membuat sebuah static factory method untuk perulangan diluar objek untuk tipe parameter yang dibutuhkan. Pola ini kbanyakan digunakan untuk function object(item 21), misalnya Collection.reverseOrder, tapi hal ini juga digunakan untuk collection, misalnya Collection.emptySet.

Misalkan kita memiliki interface yang mendeskripsikan sebuah fungsi yang menerima dan mengembalikan nilai kembalian dengan tipe T.

```
public interface UnaryFunction<T> {
        T apply(T arg);
}
```

Sekarang, bayangkan bahwa kita ingin untuk menyediakan fungsi identitas. Itu tidak akan berguna untuk membuat yang baru tiap kali dibutuhkan, sebagaimana itu stateless. Jika generic adalah abstrak, kita akan perlu sebuah fungsi identitas untuk tipe, tapi semenjak mereka telah dihapus, kita perlu hanya sebuah generic singleton. Lihat gimana itu jalan.

```
// Generic singleton factory pattern
private static UnaryFunction<Object> IDENTITY_FUNCTION =
        new UnaryFunction<Object>() {
                public Object apply(Object arg) {
                        return arg;
                }
        };
// IDENTITY_FUNCTION is stateless and its type parameter is
// unbounded so it's safe to share one instance across all types.
@SuppressWarnings("unchecked")
public static <T> UnaryFunction<T> identityFunction() {
        return (UnaryFunction<T>) IDENTITY_FUNCTION;
}
```

Casting IDENTITY_FUCNTION menjadi (UnaryFunction<T>) mengenerate sebuah unchecked cast warning, sebagai UnaryFunction<Object> adalah bukan UnaryFunction<T> untuk setiap T. Tapi, identity function adalah special. Dia mengembalikan argument atau parameter tanpa modifikasi. Sehingga kita tahu bahwa mereka aman untuk digunakan sebagai UnaryFunction<T> apapun nilai dari T. Oleh karena itu, kita bisa menggunakan suppress untuk mengatasi warning yang degenerate oleh

proses casting. Jika kita selesai melakukan ini, kode akan di compile tanpa error dan warning.

```
// Sample program to exercise generic singleton
public static void main(String[] args) {
        String[] strings = { "jute", "hemp", "nylon" };
        UnaryFunction<String> sameString = identityFunction();
        for (String s : strings)
                System.out.println(sameString.apply(s));
        Number[] numbers = { 1, 2.0, 3L };
        UnaryFunction<Number> sameNumber = identityFunction();
        for (Number n : numbers)
                System.out.println(sameNumber.apply(n));
}
```

Berikutnya akan berlanjut tentang, recursive type bound untuk membuat proses compare nilai dengan interface comparable menjadi lebih baik, efektif dan berkualitas. Contohnya dapat kita saksikan di bawah ini.

```
// Using a recursive type bound to express mutual comparability
public static <T extends Comparable<T>> T max(List<T> list) {...}
```

sebagaimana dari fungsi compare yang hanya bisa dilakukan untuk tipe data yang sama, maka method di atas bertujuan yang sama pula, kita akan mencari nilai maximum dari kumpulan data yang bertipe sama, dan system juga akan mengenali secara otomatis tipenya, ketika parameter yang di passing itu mulai dicompile.

```
// Returns the maximum value in a list - uses recursive type bound
public static <T extends Comparable<T>> T max(List<T> list) {
        Iterator<T> i = list.iterator();
        T result = i.next();
        while (i.hasNext()) {
                T t = i.next();
                if (t.compareTo(result) > 0)
                        result = t;
                }
```

```
        return result;
}
```

Sederhananya, generic method lebih aman dan lebih mudah untuk digunakan, ketimbang raw type atau raw method yang membutuhkan proses casting value di sisi client tiap kali input dan output nilai dimanipulasi dan akan diperoleh. Sebagaimana dengan tipe data, kita harus memastikan bahwa method yang diciptakan tidak memerlukan casting value atau variable di dalamnya. Yang terakhit kita harus memastikan untuk membuat method yang baru diciptakan bisa exist tanpa merusak kinerja dari method yang existing.

Item 28. Gunakan Bounded Wildcard untuk meningkatkan fleksibilitas dari API.

Bounded wildcard bertujuan untuk membuat sebuah API menjadi fleksibel dan dapat diakomodasi oleh banyak kondisi. Sehingga kita tidak secara spesifik menentukan tipe data dari sekelompok collection atau sejenisnya, kita hanya menyebut dia secara umum dan harus besar saja. Misalnya, sebuah collection yang dideklarasikan seperti, List<String>, hanya bisa diIsi dengan nilai String saja, apabila dipaksa untuk tipe lainnya, maka akan terjadi runtime error, dan ini menjadi masalah, seandainya kita tidak hanya akan menyimpan nilai berupa String saja, dan tidak tahu persis selain String apa saja yang akan disimpan. Contoh berikutnya, List<Object>, Kita bisa menyimpan berbagai bentuk nilai di sini, karena semua objek tipe data merupakan turunan dari kelas object. Jadi sederhananya gimana kita membuat objek itu merupakan variable yang fleksibel untuk dimanipulasi dalam koleksi data. Karena kita hanya bisa menyimpan input nilai yang merupakan object dengan tipe data yang sama atau turunannya, seperti String yang merupakan turunan dari Object.

Kadang-kadang kita butuh lebih fkelsibel daripada tipe spesifik yang disediakan. Perhatikan di-bawah ini adalah API public.

```
public class Stack<E> {
        public Stack();
        public void push(E e);
        public E pop();
        public boolean isEmpty();
}
```

Suppose we want to add a method that takes a sequence of elements and pushes them all onto the stack. Here's a first attempt:

```
// pushAll method without wildcard type - deficient!
public void pushAll(Iterable<E> src) {
for (E e : src)
        push(e);
}
```

Bayangkan kita ingin menambahkan sebuah method yang mengambil elemen secara sekuensial dan di-insertkan semua-nya ke dalam Stack.

```
// pushAll method without wildcard type - deficient!
public void pushAll(Iterable<E> src) {
        for (E e : src)
                push(e);
}
```

Method di-compile dengan baik, tapi secara keseluruhan belum begitu baik. Jika tipe elemen dari Iterable src persis sama dengan Stack, kode program akan jalan dengan baik. Tapi, jika kita memiliki sebuah Stack<Number> dan kita memanggil push(intVal), dimana intVal tipenya adalah integer. Kode akan jalan dengan baik, karena integer merupakan sub-type dari Number. Sehingga, logikanya kode program seharusnya jalan dengan baik, juga :

```
Stack<Number> numberStack = new Stack<Number>();
Iterable<Integer> integers = ... ;
numberStack.pushAll(integers);
```

Jika kita coba code di-atas, kita akan mendapatkan error message karena, sebagaimana catatan di-atas, parameter hanya spesifik satu jenis nilai saja, tidak semua tipe bisa match.

```
StackTest.java:7: pushAll(Iterable<Number>) in Stack<Number>
cannot be applied to (Iterable<Integer>)
numberStack.pushAll(integers);
```

Untungnya, ada jalan keluarnya, java menyediakan jenis parameter spesial yang disebut bounded wildcard type untuk meng-antisipasi kondisi tersebut. Tipe yang menjadi input untuk sebagai parameter pada method pushAll seharusnya tidak Iterable E, tapi Iterable extends E, dan ada wildcard type yang artinya persis : Iterable<? Extends E>.

```
// Wildcard type for parameter that serves as an E producer
public void pushAll(Iterable<? extends E>src) {
        for (E e : src)
                push(e);
}
```

Dengan mengubah kode program, tidak hanya Stack berhasil ketika di-compile, tapi juga terjadi pada deklarasi original pushAll pada kode client. Karena Stack dan hasil compile clientnya sama-sama baik, kita tahu bahwa semua-nya merupakan tipe yang aman. Sekarang jika kita ingin membuat method popAll yang menghapus tiap elemen dari Stack dan menambahkan elemen pada collection. Perhatikan bawah ini gimana method popAll terlihat.

```
// popAll method without wildcard type - deficient!
public void popAll(Collection<E> dst) {
        while (!isEmpty())
                dst.add(pop());
}
```

Lagi, hasil compile ini baik dan jalan dengan baik jika tipe element dari collection tujuan persis sama dengan Stack. Tapi lagi-lagi, itu tidak terlihat tepat secara keseluruhan. Bayangkan jika kita memiliki Stack<Number> dan

tipe dari variable adalah objek. Jika kita mengambil element dari Stack dan menyimpan itu dalam sebuah variable tertentu, itu di-compile dan jalan tanpa error, sehingga tidak seharusnya kita bisa melakukan ini.

```
Stack<Number> numberStack = new Stack<Number>();
Collection<Object> objects = ... ;
numberStack.popAll(objects);
```

Jika kita mencoba compile code client lagi dengan versi method popAll di-atas, kita akan mendapatkan error yang sama dengan yang pernah kita dapatkan pada versi pertama method pushAll, Collection<Object> adalah bukan sub-type dari Collection<Number>. Sekali lagi, tipe wildcard menyediakan jalan keluar. Tipe untuk parameter input dapa method popAll tidak seharusnya Collection E, tapi Collection yang super-classnya adalah E. Lagi, ada tipe wildcard yang memiliki arti sama dengan itu : Collection<? super E>. Mari modifikasi method popAll.

```
// Wildcard type for parameter that serves as an E consumer
public void popAll(Collection<? super E>dst) {
        while (!isEmpty())
                dst.add(pop());
}
```

PECS stands for producer-extends, consumer-super.

Dengan kata lain, jika sebuah parameter berperan sebagai T producer, gunakan **<? Extends T>**, jika itu berperan sebagai T consumer, gunakan **<? Super T>.** Pada contoh Stack di-atas, parameter src yang berperan sebagai input untuk method pushAll, yang menciptakan instance E untuk digunakan oleh Stack. Sehingga tipe yang tepat untuk src adalah Iterable<? Extends E>, parameter dst pada method popAll berperan sebagai output yang mendapatkan instance E dari Stack, mengambil instance E dari Stack, sehingga tipe yang tepat untuk dst adalah Collection<? super E>. Mnemonic membawa prinsip mendasar yang mengarahkan pengunaan tipe wildcard.

Naftalin dan Wadler menyebut itu Get and Put principle. Maksdunya, get untuk Consumer dan put untuk producer.

static <E> E reduce(List<E> list, Function<E> f, E initVal)

Walaupun list bisa berperan sebagai keduanya, consumer dan producer, method reduce menggunakan parameter list hanya sebagai sebuah producer E. Sehingga deklarasinya seharusnya menggunakan tipe wildcard yang extends E. Parameter f menjabarkan sebuah fungsi yang keduanya berperan sebagai instance producer dan consumer untuk E. Sehingga tipe wildcard akan tidak tepat untuk itu. Ini hasil deklarasi-nya method.

// Wildcard type for parameter that serves as an E producer
*static <E> E reduce(**List<? extends E>**list, Function<E> f, E initVal)*
Dan, akankah perubahan ini membuat sebuah perbedaan pada prakteknya ? sepertinya benar. Jika kita memiliki sebuah List<Integer>, dan kita ingin reduce itu dengan sebuah Function<Number>. Ini tidak akan di-compile dengan deklarasi originalnya, tapi itu dilakukan, saat kita menambahkan tipe bounded wildcard.

Now let's look at the union method from Item 27. Here is the declaration:
public static <E> Set<E> union(Set<E> s1, Set<E> s2)

Both parameters, s1 and s2, are E producers, so the PECS mnemonic tells us that the declaration should be:
public static <E> Set<E> union(Set<? extends E> s1, Set<? extends E> s2)

Catat bahwa tipe return adalah masih Set<E>. Jangan menggunakan tipe wildcard sebagai return value. Daripada menyediakan fleksibilitas tambahan untuk pengguna, itu akan memaksa mereka untuk menggunakan tipe wildcard pada client code. Digunakan sepantasnya, tipe wildcard hampir tidak terlihat untuk kelas-nya client. Mereka menyebabkan method untuk menerima parameter yang seharusnya mereka diterima dan mereka seharusnya menolak yang seharusnya ditolak. Jika pengguna kelas harus

berpikir tentang tipe wildcard, ada kemungkinan sesuatu yang salah terjadi dengan API di kelas tertentu.

Sayangnya, tipe inference sedikit kompleks. Lihat revisi deklarasi untuk method union, kita mungkin berpikir yang kita bisa lakukan.

Set<Integer> integers = ... ;
Set<Double> doubles = ... ;
Set<Number> numbers = union(integers, doubles);

If you try it you'll get this error message:
Union.java:14: incompatible types found : Set<Number & Comparable<? extends Number &
Comparable<?>>>
required: Set<Number>
Set<Number> numbers = union(integers, doubles);

Untungnya, ada cara untuk mengatasi error tersebut. Jika compiler bukan tipe inference yang kita harapkan di-miliki, kita bisa mengetahui apa tipe yang bisa digunakan dengan sebuah tipe parameter eksplisit. Ini bukan sesuatu yang kita harus lakukan sangat sering, yang mana adalah sebuah hal yang baik, sebagaimana tipe parameter eksplisit tidak terlalu baik. Dengan tambahan tipe parameter eksplisit, program akan di-compile dengan baik.

Set<Number> numbers = Union.<Number>union(integers, doubles);

Next let's turn our attention to the max method from Item 27. Here is the original declaration:
public static <T extends Comparable<T>> T max(List<T> list)

Here is a revised declaration that uses wildcard types:
public static <T extends Comparable<? super T>> T max(List<? extends T>list)

Untuk mendapatkan revisi deklarasi dari originalnya, kita menerapkan PECS sebanyak 2 kali. Aplikasi sederhana untuk parameter list. Dia menciptakan

instance T, sehingga kita bisa mengubah tipe dari List<T> menjadi List<? extends T>. Aplikasi yang sedikit tricky adalah parameter denga tipe T. Ini adalah pertama kali kita melihat sebuah tipe wildcard diterapkan untuk tipe dari parameter. T aslinya special hanya untuk extends Comparable<T>, tapi sebuah comparable dari T menggunakan instance T (dan menciptakan nilai integer sebagai indikasi adanya hubungan). Oleh karena itu, tipe tipe parameter dari Comparable<T> digantikan oleh tipe bounded wildcard Comparable<? super T>. Comparable selalu menjadi consumer, sehingga kita seharusnya menggunakan Comparable<? super T> sebagai ganti dari Comparable<T>. Sama juga dengan comparator, sehingga kita seharusnya selalu menggunakan Comparator<? super T> sebagai ganti dari Comparator<T>.

Deklarasi method max mungkin merupakan yang paling kompleks dalam satu buku ini. Apakah semakin kompleks membuat kita memperoleh sesuatu ? iya/ Ini dia contoh sederhana dari list yang akan dikecualikan oleh deklarasi originalnya tapi dibolehkan oleh revisinya.

List<ScheduledFuture<?>> scheduledFutures = ... ;.

Alasan yang membuat kita tidak bisa menggunakan deklarasi original method untuk list ini adalah *java.util.concurrent.ScheduledFuture* tidak meng-implement Comparable<ScheduledFuture>. Sebaliknya, merupakan sub-interface dari Delayed, yang meng-extends Comparable<Delayed>. Dengan kata lain, sebuah instance ScheduledFuture tidak hanya comparable untuk instance ScheduledFuture lainnya. Itu juga comparable untuk setiap instances Delayed, dan itu cukup untuk deklarasi originalnya menolak untuk tidak bisa dilakukan.

Ada sedikit masalah dengan deklarasi revisi dari method max. Itu mencegah method untuk tidak bisa di-compile.

// Won’t compile - wildcards can require change in method body!

```
public static <T extends Comparable<? super T>> T max( List<? extends T> list)
{
        Iterator<T> i = list.iterator();
        T result = i.next();
        while (i.hasNext()) {
                T t = i.next();
                if (t.compareTo(result) > 0)
                        result = t;
        }
        return result;
}
```

Here's what happens when you try to compile it:

```
Max.java:7: incompatible types found : Iterator<capture#591 of ? extends T>
required: Iterator<T>
Iterator<T> i = list.iterator();
```

Revisinya,

```
Iterator<? extends T> i = list.iterator();
```

Itulah perubahan yang kita harus lakukan pada kode program.

Ada satu lagi topic wildcard yang masih berkaitan yang akan kita diskusikan. Ada sebuah dualitas antara tipe parameter dan wildcard, dan banyak method bisa dideklarasikan menggunakan satu atau lainnya. Contohnya, ada dua kemungkinan deklarasi untuk method static untuk ditukar 2 index item pada list. Pertama menggunakan unbounded type parameter(item 27) dan yang kedua menggunakan unbounded wildcard.

```
// Two possible declarations for the swap method
public static <E> void swap(List<E> list, int i, int j);
public static void swap(List<?> list, int i, int j);
```

Yang mana dari keduanya yang lebih disukai, dan kenapa ? dalam API yang bersifat public, yang kedua lebih baik karena lebih simple. Kita lewatkan dalam sebuah list, list apapun, dan method menukar index dari elementnya. Tidak adak yang perlu dikhawatirkan dengan tipe parameter. Sebagaimana

sebuah aturan, jika tipe parameter muncul hanya sekali dalam sebuah deklarasi method, gantikan itu dengan sebuah wildcard. Jika itu adalah sebuah unbounded type parameter, gantikan itu dengan sebuah bounded wildcard. Ada satu masalah dengan deklarasi kedua untuk method swap. Yang mana menggunakan sebuah wildcard menggantikan sebuah tipe parameter. Implementasi tidak akan di-compile.

```
public static void swap(List<?> list, int i, int j) {
        list.set(i, list.set(j, list.get(i)));
}
```

Trying to compile it produces this less-than-helpful error message:

```
Swap.java:5: set(int,capture#282 of ?) in List<capture#282 of ?> cannot be applied to (int,Object)
list.set(i, list.set(j, list.get(i)));
```

Itu tidak terlihat benar, bahwa kita tidak bisa meng-insert kembali element ke dalam list yang telah kita retrieve. Masalahnya adalah tipe dari list adalah List<?>, dan kita tidak bisa meng-insert nilai baru apapu kecuali null ke dalam sebuah List<?>. Sayangnya. Ada sebuah jalan untuk meng-implementasikan method tanpa beralih menjadi casting yang tidak aman atau tipe raw. Idenya adalah untuk membuat sebuah method helper untuk capture tipe wildcard. Method helper harus bersifat generic, method di-perintahkan untuk capture tipenya.

```
public static void swap(List<?> list, int i, int j) {
        swapHelper(list, i, j);
}
// Private helper method for wildcard capture
private static <E> void swapHelper(List<E> list, int i, int j) {
        list.set(i, list.set(j, list.get(i)));
}
```

Method swapHelper mengetahui bahwa list adalah List<E>. Oleh karena itu, dia tahu bahwa setiap nilai yang di-retrieve dari list adalah bertipe E, dan itu aman untuk meng-insertkan nilai apapun dengan tipe E ke dalam list. Dia

mengizinkan kita untuk –mem-public-kan deklarasi dari method swap yang dasarnya adalah wildcard, ketika memperoleh keuntungan atas kompleksitas dari method generic secara internal. Pengguna dari method swap tidak harus menghadapu deklarasi method swapHelper yang lebih kompleks, tapi mereka memperoleh keuntungan dari itu.

Bounded WildCard menganut istilah PECS, yaitu producer-extends, consumer-super. Kaitan ini dengan deklarasi dari parameter input dan output sebuah manipulasi nilai, contohnya adalah di bawah ini.

Input-Producer- List<? extends E>, maksudnya adalah nilai input merupakan list yang contentnya adalah semua object turunan dari E.

Output - Consumer - List<? Super T>, nilai output merupakan list yang contentnya adalah parent dari T.

Berikutnya, Comparable dan Comparator merupakan jenis consumer, sehingga berlaku Aturan kedua, outputnya merupakan object dengan parent T. Contoh deklarasinya adalah T extends Comparable<? super T>, T implements Comparator<? super T>.

Simpulannya, menggunakan tipe wildcard dalam API kita, sebenarnya rumit, membuat API jauh lebih fleksibel. Jika kita membuat sebuah library yang akan memperluas pengunaannya, ketepatan penggunaan tipe wildcard seharusnya benar-benar dipertimbangkan sebagai hal yang diharuskan. Ingat aturan dasar, producer extends, consumer super (PECS). Dan ingat bahwa semua Comparable dan Comparator adalah consumer.

Item 29. Consider Type safe heterogenous containers.

Umumnya, kebanyakan penggunaan Generic adalah untuk Collections, antara lain Set dan Map, dan Single-Element Container, misalnya ThreadLocal dan AtomicReference. Dalam semua penggunaan itu, mereka adalah Container yang menggunakan parameter. Ini membatasi kita

terhadap jumlah yang tetap untuk jenis parameter per container. Normalnya, itu sebenarnya yang tepatnya kita inginkan. Sebuah Set memiliki jenis parameter tunggal, merepresentasikan jenis element-nya, sebuah Map memiliki 2, merepresentasikan kunci-nya dan jenis nilainya.
Terkadang, bagaimanapun kita perlu fleksibilitas yang lebih. Contohnya, sebuah baris pada database bisa memiliki banyak kolom nilai, dan akan baik untuk bisa mengakses semuanya dengan tipe yang aman. Untungya, ada sebuah cara yang mudah untuk mendapatkan efek ini. Idenya adalah untuk menjadikan kunci sebagai parameter bukannya container. Kemudian, menggunakan key untuk container untuk meng-insert atau me-retrieve sebuah nilai. Jenis sistem Generic digunakan untuk jaminan bahwa jenis dari nilai cocok dengan key-nya.

Sebagai contoh dari pendekatan ini, pertimbangkan sebuah Favorites yang mengizinkan client-nya untuk menyimpan dan me-retrieve sebuah instance favorite dari banyak kelas lain-nya. Objek Class akan memainkan peran dari parameter key. Alasan kenapa ini bisa jalan adalah bahwa kelas Class telah di-generified dalam release java 5. Tipe dari kelas literal tidak lagi menggunakan Class, tapi Class<T>. Contohnya, String.class adalah bagian dari tipe Class<String>, dan Integer.class adalah bagian dari Class<Integer>. Ketika sebuah kelas literal dilewatkan diantara method untuk meng-komunikasikan informasi untuk compile time dan runtime , mereka disebut tipe token.

API untuk kelas Favorites simple. Itu mirip seperi Map tunggal, kecuali kuncinya adalah parameter bukannya Map. Client memiliki sebuah objek Class ketika setting dan getting Favorites. Ini API-nya.

```
// Typesafe heterogeneous container pattern - API
public class Favorites {
        public <T> void putFavorite(Class<T> type, T instance);
        public <T> T getFavorite(Class<T> type);
}
```

Ini adalah program sederhana sebagai latihan kelas Favorites, menyimpan, me-retrieve dan, me-print sebuah instance favorite String, Integer dan Class.

```
// Typesafe heterogeneous container pattern - client
public static void main(String[] args) {
        Favorites f = new Favorites();
        f.putFavorite(String.class, "Java");
        f.putFavorite(Integer.class, 0xcafebabe);
        f.putFavorite(Class.class, Favorites.class);

        String favoriteString = f.getFavorite(String.class);
        int favoriteInteger = f.getFavorite(Integer.class);
        Class<?> favoriteClass = f.getFavorite(Class.class);
        System.out.printf("%s %x %s%n", favoriteString,
        favoriteInteger, favoriteClass.getName());
}
```

Sebagaimana yang kita harapkan, program ini mencetak Java Cafebabe Favorites. Sebuah instance Favorites adalah typesafe. Itu tidak akan mengembalikan sebuah nilai Integer ketika kita meminta untuk nilai String. Itu juga heterogeneous, tidak seperti map normal, semua key memiliki tipe yang berbeda-beda. Oleh karena itu, kita menyebut Favorites adalah sebuah typesafe heterogeneous container.

Implementasi dari favorites sedikit mengejutkan, kodenya tidak terlalu panjang.

```
// Typesafe heterogeneous container pattern - implementation
public class Favorites {
        private Map<Class<?>, Object> favorites = new HashMap<Class<?>,
Object>();

        public <T> void putFavorite(Class<T> type, T instance) {
                if (type == null)
                        throw new NullPointerException("Type is null");
                favorites.put(type, instance);
        }
```

```
    public <T> T getFavorite(Class<T> type) {
        return type.cast(favorites.get(type));
    }
}
```

Tiap instance dari favorites terbuah dari sebuah Map<Class<?>, Object> bersifat private yang disebut favorites. Kita mungkin berpikir bahwa kita tidak bisa meletakkan segala sesuatu ke dalam Map ini karena unbounded wildcard type, tapi kebenaran-nya cukup berlawanan. Tujuannya untuk memberitahukan bahwa wildcard type bersarang, itu bukan jenis dari Map yang berupa sebuah tipe wildcard tapi tipe dari key-nya. Ini artinya bahwa setiap key bisa memiliki tipe parameter yang berbeda-beda, yang pertama bisa Class<String>, selanjutnya Class<Integer>, dan seterusnya. Itu adalah dimana heterogeneity berasal.

Hal berikutnya adalah untuk memberitahukan bahwa jenis nilai dari favorites Map merupakan objek yang sederhana. Dengan kata lain, Map bukan jaminan atas hubungan dari key dan value, yang mana bahwa setiap value memiliki tipe yang sama dengan key. Faktanya, java tidak cukup powerful untuk meng-ekspresikan ini. Tapi, kita tahu bahwa itu benar, dan kita memiliki keuntungan ketika saat akan retrieve sebuah favorites.

Implementasi dari method putFavorite adalah trivial, hal ini sesederhana meletakkan sebuah objek Class tertentu untuk di-mapping ke dalam instance favorites. Sebagai catatan, ini melanggar proses "link" antara key dan value-nya. tapi bisa dimaklumi, karena method getFavorites bisa dan melakukan penciptaan ulang "link" ini.

Implementasi terhadap method getFavorites beresiko daripada method putFavorites. pertama, kita memperoleh nilai dari Map favorites dan di hubungkan dengan objek Class tertentu. ini merupakan cara terbaik untuk mengembalikan reference objek sebagai nilai kembalian/return value, tapi merupakan jenis compile yang salah. Jenis ini sederhana-nya adalah sebuah

objek (jenis nilai dari Map favorites) dan kita perlu untuk kembalikan nilai-nya dengan sebuah T. Sehingga, method getFavorites melakukan casting terhadap reference dari nilai tersebut menjadi Class object, menggunakan method cast dari Class.

method cast adalah operator casting java yang dinamis. mereka melakukan pengecekan terhadap parameter apakah merupakan jenis dari Class object. jika benar, dia akan mengembalikan nilainya, sebaliknya akan men-throws sebuah ClassCastException. kita tahu bahwa pemanggilan method casting pada method getFavorite tidak akan pernah mendapatkan

ClassCastException, assumsikan client code di-compile dengan aman. selanjutnya katakanlah, kita tahu bahwa nilai di-dalam Map favorites selalu persis dengan jenis dari key-nya.

Sehingga, apakah yang method cast lakukan untuk kita, hanya mengembalikan argumen-nya atau parameter-nya saja ?

```
public class Class<T> {

        T cast(Object obj);

}
```

Ini adalah apa yang benar-benar di perlukan oleh method getFavorites. Hal ini adalah apa yang mengizinkan kita untuk membuat Favorites yang aman tanpa ada proses casting ke T yang tidak didahului oleh proses checking.

Ada 2 batasan pada kelas Favorites yang patut di-catat.

1. Client yang berbahaya bisa membuat instance dari Favorites corrupt, sederhananya membuat sebuah Class object menggunakan bentuk raw-nya. Tapi hasilnya, kode client akan menciptakan sebuah unchecked warning saat dilakukan compile. Ini tidak berbeda dari bentuk collection yang normal, semisal HashSet dan HashMap. Kita bisa dengan gampang insert sebuah String ke dalam

HashSet<Integer> dengan menggunakan bentuk raw dari HashSet. Itu mengatakan, kita bisa memiliki proses runtime yang aman jika kita ingin berkorban untuk itu. Caranya adalah dengan memastikan Favorites tidak melanggar jenis tunggal-nya adalah untuk memiliki method putFavorite yang melakukan pengecekan terhadap instances apakah reference yang direpresentasikan oleh jenis parameter itu.

```
// Achieving runtime type safety with a dynamic cast

public <T> void putFavorite(Class<T> type, T instance) {

        favorites.put(type, type.cast(instance));

}
```

ada beberapa jenis collection wrapper pada library java.util.Collections yang melakukan hal yang sama. Merekada adalah checkedSet, checkedList, checkedMap dan lainnya. static factory method menggunakan sebuah atau dua Class object ditambahkan ke dalam collection atau Map. static factory method adalah method yang generic, pastikan tipe saat compile time, jenis dari Class object dan collection adalah persis sama. contohnya, wrapper throws sebuah ClassCastException saat runtime, jike seseorang coba untuk insert Coin ke dalam Collection<Stamp>. wrapper-wrapper itu berguna untuk melacak siapa yang meng-insert jenis nilai yang salah itu.

2. Kelas Favorites tidak bisa digunakan untuk sebuah jenis non-reifiable. Dengan kata lain, kita bisa menyimpan String atau String[] tapi bukan List<String>. Jika kita mencoba menyimpan itu, program kita tidak akan bisa di-compile. Alasan-nya adalah kita tidak bisa mendapatkan sebuah Class object untuk List<String>, List<String>.class adalah syntax error, dan itu adalah hal yang bagus juga. List<String> dan List<Integer> sharing sebuah Class object tunggal, yang adalah List.class. Itu akan mendatangkan malapetaka terhadap internal dari object Favorites jika List<String>.class dan List<Integer>.class dibolehkan dan return objek reference yang sama.

Tidak ada solusi terbaik untuk batasan yang kedua. Ada sebuah teknik yang disebut dengan super type tokens yang menjauhkan dari batasan ini, tapi teknik ini memiliki batasan yang membatasi dirinya sendiri.

Jenis token yang digunakan oleh Favorites bersifat unbounded, getFavorites dan putFavorites menerima apapun objek dari Class. Terkadang kita mungkin memerlukan batasan terhadap apa saja jenis parameter yang bisa dilewatkan ke method tertentu. Ini bisa dimungkinkan dengan bounded type token, yang mana merupakan jenis token sederhana yang menempatkan sebuah bound pada apa jenis yang bisa direpresentasikan, menggunakan jenis parameter bounded atau bounded wildcard.

Annotations API membuat penggunaan yang khusus terhadap bounded type tokens. Sebagai contohnya, ini adalah method yang membaca sebuah annotation saat runtime. Method ini diperoleh dari interface AnnotatedElement, yang di-implement oleh jenis reflective yang merepresentasikan kelas, method, field, dan element lainnya.

public <T extends Annotation> T getAnnotation(Class<T> annotationType);

Parameter *annotationType* adalah *bounded type token* merepresentasikan sebuah jenis *annotation.* Method mengembalikan element *annotation* dari jenis itu, jika mereka memiliki satu atau null, jika tidak memiliki sama sekali. Esensinya, sebuah element *annotated* adalah container beraneka ragam yang aman yang *key*-nya berjenis *annotation.*

Seandainya kita memiliki sebuah objek berjenis *Class<?>* dan kita ingin melewatkan itu pada sebuah method yang mengharuskan sebuah jenis *bounded token,* misalnya *getAnnotation.* Kita bisa *casting* objek ke *Class<? extends Annotation>,* tapi *casting* ini bersifat *unchecked,* sehingga akan memberikan sebuah warning saat compile time. Untungnya, kelas *Class* menyediakan sebuah instance method yang menggunakan cara *casting* yang lebih aman dan dynamic. Method itu disebut *asSubclass,* dan itu melakukan *casting* objek *Class* yang digunakan untuk merepresentasikan sebuah

subclass dari kelas yang digunakan oleh parameter-nya atau argument-nya.Jika proses casting sukses, method mengembalikan parameter-nya atau argument-nya, jika itu gagal, akan mendapatkan *throws ClassCastException.*

Dibawah ini adalah cara menggunakan method *asSubclass* untuk membaca sebuah *annotation* tanpa tahu jenis-nya saat compile time. Method ini di-compile tanpa *error* dan *warning.*

```
// Use of asSubclass to safely cast to a bounded type token
static Annotation getAnnotation(AnnotatedElement element, String
annotationTypeName) {

        Class<?> annotationType = null; // Unbounded type token
        try {
                annotationType = Class.forName(annotationTypeName);
        } catch (Exception ex) {
                throw new IllegalArgumentException(ex);
        }

        return
element.getAnnotation(annotationType.asSubclass(Annotation.class));
}
```

Simpulannya, penggunaan *generic* yang normal, dicontohkan oleh API *Collections,* membatasi kita untuk jenis parameter yang pasti untuk tiap-tiap container. Kita bisa menyebarkan batasan dengan menempatkan jenis parameter pada *key* ketimbang *container.* Kita bisa menggunakan objek *Class* sebagai kunci untuk beberapa *container* dengan keberagaman yang aman. Sebuah objek *Class* yang menggunakan skema ini disebut *jenis token.* Kita juga bisa menggunakan custom *key.* Sebagai contoh, kita bisa memiliki sebuah jenis *DatabaseRow* yang meng-ilustrasikan diri-nya sebagai baris pada database (*container*), dan jenis umum *Column<T>*sebagai *key*-nya.

BAB 6. Enum And Annotation.

Item 30. Used Enum Instead of int constant

Gunakanlah enum untuk mendeklarasikan nilai konstanta dari tipe data integer, karena itu lebih sederhana dan efektif untuk pengelompokkan dan operasi aritmatika, misalnya, APPLE dan ORANGE.

APPLE {red, blue, green}

ORANGE {red, blue, green}

Walaupun mereka punya atribut yang sama, namun akan lebih mudah dikenali dan dioperasikan, misalnya kita bisa memastikan enum yg dilemparkan bukan miliknya Orange dan harus Apple, maka programming akan memberikan pesan error, alasannya karena bukan sekedar nilai konstanta yang diperlukan, tapi lebih dari itu, nilai konstanta dengan syarat bertipe Apple misalnya.

Contoh lainnya yang lebih kompleks, dengan enum kelas yang memiliki procedure dan operation di dalamnya, disertai dengan constructor.

```
// Enum type with data and behavior

public enum Planet {

        MERCURY(3.302e+23, 2.439e6),

        VENUS (4.869e+24, 6.052e6),

        EARTH (5.975e+24, 6.378e6),

        MARS (6.419e+23, 3.393e6),

        JUPITER(1.899e+27, 7.149e7),

        SATURN (5.685e+26, 6.027e7),

        URANUS (8.683e+25, 2.556e7),

        NEPTUNE(1.024e+26, 2.477e7);


        private final double mass; // In kilograms

        private final double radius; // In meters

        private final double surfaceGravity; // In m / s^2
```

```
        // Universal gravitational constant in m^3 / kg s^2
        private static final double G = 6.67300E-11;
        // Constructor
        Planet(double mass, double radius) {
                this.mass = mass;
                this.radius = radius;
                surfaceGravity = G * mass / (radius * radius);
        }
        public double mass() { return mass; }
        public double radius() { return radius; }
        public double surfaceGravity() { return surfaceGravity; }
        public double surfaceWeight(double mass) {
                return mass * surfaceGravity; // F = ma
        }
}
```

untuk menghubungkan antara data dengan konstanta enum adalah dengan mendeklarasikan instance value dan membuat sebuah konstruktor yang akan menerima data dan menyimpannya di dalam field. Enum bersifat immutable artinya konstan dan tidak bisa di ubah, sama seperti tipe data primitif lainnya, sehingga setiap variable is final. Mereka bisa bersifat public, tapi lebih baik membuat mereka private dan membuat sebuah aksessor yang bersifat public seperti konstruktor dan procedure.

Contoh di atas menjelaskan bahwa konstruktor juga melakukan operasi perhitungan terhadap tingkat gravitasi, tapi itu hanya sebuah optimasi. Tingkat gravitasi sebenarnya dapat dihitung untuk setiap planet dengan menggunakan procedure surfaceWeight(), yang akan menggunakan massa

dari objek planet dan memberikan nilai dari tingkat gravitasi yang bersifat konstan. Meskipun enum Planet lebih simple, tapi dia lebih powerful. Contoh di bawah ini adalah cara menggunakan planet enum yang powerful untuk mendapatkan tingkat grativasi untuk setiap Planet.

```
public class WeightTable {

    public static void main(String[] args) {

        double earthWeight = Double.parseDouble(args[0]);

        double mass = earthWeight / Planet.EARTH.surfaceGravity();

        for (Planet p : Planet.values())

            System.out.printf("Weight on %s is %f%n", p,
    p.surfaceWeight(mass));

    }

}
```

Jangan lupa bahwa Planet seperti semua kelas enum lainnya, memiliki static procedure values(), yang mengembalikan seluruh nilai yang dideklarasikan di dalam kelas enum tersebut. Juga memiliki toString() method yang memberikan nama dari tiap-tiap euum konstan yang dideklarasikan, di atas berupa nama-nama planet. Itu membuat mereka mudah untuk di print ke layar, seandainya kita tidak suka dengan apa yang di tampilkan oleh toString(), kita bisa melakukan override, dan kustomisasi isi dari method itu berdasarkan keinginan kita sendiri.

Di bawah adalah hasil dari main class di atas ketika kita mengexecute kode itu.

```
Weight on MERCURY is 66.133672

Weight on VENUS is 158.383926

Weight on EARTH is 175.000000
```

Weight on MARS is 66.430699

Weight on JUPITER is 442.693902

Weight on SATURN is 186.464970

Weight on URANUS is 158.349709

Weight on NEPTUNE is 198.846116

Contoh di atas adalah contoh enum kelas dengan constanta behaviour yang sama, menghitung gravitasi, bagaimana jikalau mereka, tiap2 konstanta memiliki behaviour yang berbeda. Perhatikan contoh di bawah ini, bandingkan codenya.

```
// Enum type that switches on its own value - questionable
public enum Operation {
    PLUS, MINUS, TIMES, DIVIDE;
    // Do the arithmetic op represented by this constant
    double apply(double x, double y) {
        switch(this) {
            case PLUS: return x + y;
            case MINUS: return x - y;
            case TIMES: return x * y;
            case DIVIDE: return x / y;
        }
        throw new AssertionError("Unknown op: " + this);
    }
}
```

code ini jalan dengan baik, tetapi terlihat tidak efisien dan kurang tepat. Code tidak akan melakukan compile tanpa statemen throw di akhir method. Meskipun code mungkin akan jalan dengan baik. Lebih buruk lagi, code di atas rapuh, jikalau kita menambahkan sebuah operasi baru, tetapi lupa untuk menambahkannya di dalam switch, maka kita akan mengalami kegagalan ketika kita akan menggunakan operasi yang baru itu. Tetapi kita memiliki alternative lain yang lebih baik, declare abstract method in enum, and override that method for each konstanta enum, perhatikan code di bawah ini.

```
// Enum type with constant-specific method implementations

public enum Operation {

        PLUS { double apply(double x, double y){return x + y;} },

        MINUS { double apply(double x, double y){return x - y;} },

        TIMES { double apply(double x, double y){return x * y;} },

        DIVIDE { double apply(double x, double y){return x / y;} };

        abstract double apply(double x, double y);

}
```

pada jenis enum yang kedua, sangat sial jika kita sampai lupa untuk menambahkan operation baru dan lupa override apply method, karena tiap constanta diikuti oleh implement method yang menghandlenya. Compiler juga akan mengingatkan kita jikalau kita lupa, karena kita wajib implement abstract method untuk semua constanta yang di declare di dalam enum.

Konstanta spesific method bisa dikombinasikan dengan konstanta spesific data. Lihat Contoh di bawah ini.

```
// Enum type with constant-specific class bodies and data

public enum Operation {
```

```
        PLUS("+") {
            double apply(double x, double y) { return x + y; }
        },
        MINUS("-") {
            double apply(double x, double y) { return x - y; }
        },
        TIMES("*") {
            double apply(double x, double y) { return x * y; }
        },
        DIVIDE("/") {
            double apply(double x, double y) { return x / y; }
        };
        private final String symbol;
        Operation(String symbol) { this.symbol = symbol; }
        @Override public String toString() { return symbol; }
        abstract double apply(double x, double y);
}
```

Untuk beberapa kasus, method toString() sangat berguna untuk menjelaskan details kejadian dalam bentuk tulisan atau statemen. Lihat contoh code di bawah ini.

```
public static void main(String[] args) {
        double x = Double.parseDouble(args[0]);
        double y = Double.parseDouble(args[1]);
```

```
        for (Operation op : Operation.values())
            System.out.printf("%f %s %f = %f%n", x, op, y, op.apply(x, y));
}
```

Jalankan program dengan nilai x=2 dan y=4. Hasilnya,

```
2.000000 + 4.000000 = 6.000000
2.000000 - 4.000000 = -2.000000
2.000000 * 4.000000 = 8.000000
2.000000 / 4.000000 = 0.500000
```

enum type memiliki valueOf(String) method yang otomatis mengenerate nama konstanta menjadi konstanta itu sendiri. Jikalau toString() telah di override, pertimbangkan untuk menulis fromString() method, gunanya untuk translate custom string we created to enum yang bersangkutan. Lihat contoh code di bawah ini.

```
// Implementing a fromString method on an enum type
private static final Map<String, Operation> stringToEnum = new HashMap<String, Operation>();
static { // Initialize map from constant name to enum constant
        for (Operation op : values())
            stringToEnum.put(op.toString(), op);
}
// Returns Operation for string, or null if string is invalid
public static Operation fromString(String symbol) {
        return stringToEnum.get(symbol);
}
```

Perhatikan, static method akan dieksekusi sesaat ketika objek stringToEnum diciptakan, membuat masing2 konstanta menjadi bagian dari map dengan menggunakan konstruktor akan menyebabkan error saat compile. Ini adalah hal yang baik, karena itu akan menyebabkan NullPointerException jika itu diperbolehkan. Konstruktor enum tidak di-izinkan untuk mengakses masing-masing static enum yang di declare didalamnya, kecuali untuk constan field di waktu compile. Ini diperlukan karena static fields belum dibuatkan instance ketika konstruktor dijalankan.

Kekurangan dari constanta dengan spesific method adalah akan terasa lebih sulit untuk share code diantar semua enum konstanta. Contohnya, bayangkan sebuah enum yang isinya hari dalam satu pekan in dalam paket payroll. Enum ini memiliki kemampuan untuk menghitung jumlah penghasilan pekerja per jamnya dan jumlah jam mereka kerja dalam hari yang sama. Dalam lima hari kerja mereka bekerja seperti biasa waktu normal, 2 hari di akhir pekan mereka bekerja dengan label lembur/overtime. Dengan menggunakan switch statement, lebih mudah untuk melakukan kalkulasi dengan menerapkan multiple case untuk kedua perhitungan di atas. Perhatikan code di bawah ini.

```
// Enum that switches on its value to share code - questionable
enum PayrollDay {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY;
    private static final int HOURS_PER_SHIFT = 8;
    double pay(double hoursWorked, double payRate) {
        double basePay = hoursWorked * payRate;
        double overtimePay; // Calculate overtime pay
        switch(this) {
        case SATURDAY: case SUNDAY:
```

```
                overtimePay = hoursWorked * payRate / 2;
            default: // Weekdays
                overtimePay = hoursWorked <= HOURS_PER_SHIFT ?
                0 : (hoursWorked - HOURS_PER_SHIFT) * payRate / 2;
                break;
                }
            return basePay + overtimePay;
        }
    }
```

Code yang kita gunakan di atas tidaklah dilarang, tetapi akan sangat meragukan dari segi perawatan atau maintenance code. Bayangkan kamu menambahkan elemen baru di enum, mungkin special value untuk melambangkan hari libur, tetapi kamu lupa untuk mengaitkannya dengan switch statemen. Aplikasi akan tetap jalan, tapi diam-diam aplikasi akan menghitung jumlah pembayaran yang sama untuk pekerja, antar hari libur dan hari kerja normal.

Untuk memperbaiki performa dari keakuratan kalkulasi, maka.

1. Duplikasi kalkulasi overtime untuk tiap-tiap kostanta enum.
2. Pindahkan ke 2 method pembantu untuk kalkulasi, 1 untuk hitung pada hari kerja, 1 lagi untuk hitung akhir pekan. Dan eksekusi masing-masing method dari tiap konstantan enum.

Cara-cara ini sebenarnya mengurangi readability dan menngkatkan peluang untuk mengalami error.

Dibawah ini code yang lebih baik.

```
// The strategy enum pattern

enum PayrollDay {
```

```
    MONDAY(PayType.WEEKDAY), TUESDAY(PayType.WEEKDAY),
    WEDNESDAY(PayType.WEEKDAY), THURSDAY(PayType.WEEKDAY),
    FRIDAY(PayType.WEEKDAY), SATURDAY(PayType.WEEKEND),
SUNDAY(PayType.WEEKEND);

    private final PayType payType;
    PayrollDay(PayType payType) { this.payType = payType; }

    double pay(double hoursWorked, double payRate) {
        return payType.pay(hoursWorked, payRate);
    }

    // The strategy enum type
    private enum PayType {
        WEEKDAY {
            double overtimePay(double hours, double payRate) {
                return hours <= HOURS_PER_SHIFT ? 0 :
                (hours - HOURS_PER_SHIFT) * payRate / 2;
            }
        },
        WEEKEND {
            double overtimePay(double hours, double payRate) {
            return hours * payRate / 2;
        }
```

```
        };
        private static final int HOURS_PER_SHIFT = 8;
        abstract double overtimePay(double hrs, double payRate);
        double pay(double hoursWorked, double payRate) {
                double basePay = hoursWorked * payRate;
                return basePay + overtimePay(hoursWorked, payRate);
                }
        }
}
```

Switch statement tidak cocok untuk enum yang berada di bawah control kita, artinya Switch tidak cocok untuk enum yang kita buat dan kita operasikan sendiri, switch akan berguna untuk enum yang dibuat orang lain, dan kita berharap untuk menggunakan fungsinya, dan enum itu memiliki constant behaviour, tidak memiliki banyak kemungkinan untuk dioperasikan. Contohnya seperti di bawah ini.

```
// Switch on an enum to simulate a missing method
public static Operation inverse(Operation op) {
        switch(op) {
                case PLUS: return Operation.MINUS;
                case MINUS: return Operation.PLUS;
                case TIMES: return Operation.DIVIDE;
        case DIVIDE: return Operation.TIMES;
        default: throw new AssertionError("Unknown op: " + op);
        }
}
```

Enum secara umum berbicara tentang perbandingan performance daripada menggunakan konstanta integer. Kekurangan enum dibandingkan dengan integer konstanta hanya waktu dan memory yang digunakan lebih banyak ketika akan melakukan initialize tipe enum.

Kapan menggunakan enum ? kapan saja ketika anda memerlukan kumpulan konstanta yang sudah fixed. Termasuk pengetahuan umum, seperti Planet, Hari, Pion Catur dan lain lain. Termasuk juga variable yang anda ketahui berupa konstanta saat akan mengcompile aplikasi, seperti pilihan menu, command line flags, operation codes and others.

Akhir kata, keuntungan dari menggunakan enum type dibandingkan dengan konstanta integer adalah, enum jauh lebih mudah di baca dan dimengerti, lebih aman, dan lebih powerful. Banyak enum type tidak membutuhkan explicit konstruktor dan member lainnya, tetapi banyak keuntungan pula dengan meng-associate data dengan tiap-tia konstanta enum dengan menggunakan method yang behaviornya memberi efek pada data.

Jauh lebih sedikit keunggulan enum dari associate multiple behaviour dengan method tunggal. Dalam kasus yang sebenarnya agak aneh, lebih baik dengan konstanta spesifik method yang mengganti nilai dari diri mereka sendiri. Pertimbangkan, strategy enum pattern jika banyak konstanta enum yang menggunakan behavior yang sama.

Item 30. Used Enum Instead of ordinal - Gunakan enum selain ordinals.

Banyak tipe enum yang secara natural di-associate dengan nilai integer tunggal. Setiap enum memiliki sebuah ordinal() method, yang mengembalikan posisi index dari tiap-tiap konstanta enum di dalam tipe enum itu.

```
// Abuse of ordinal to derive an associated value - DON'T DO THIS

public enum Ensemble {
```

```
    SOLO, DUET, TRIO, QUARTET, QUINTET, SEXTET, SEPTET, OCTET, NONET,
DECTET;

    public int numberOfMusicians() { return ordinal() + 1; }

}
```

saat enum telah dijalankan, itu adalah mimpi buruk untuk perawatan code. Jikalau konstanta enum di re-ordered, numberOfMusician method akan rusak.

Simple solution problem : jangan membiarkan sebuah nilai di associate dengan enum dengan menggunakan ordinalnya, simpan itu dalam sebuah instance field.

```
public enum Ensemble {

    SOLO(1), DUET(2), TRIO(3), QUARTET(4), QUINTET(5),

    SEXTET(6), SEPTET(7), OCTET(8), DOUBLE_QUARTET(8),

    NONET(9), DECTET(10), TRIPLE_QUARTET(12);

    private final int numberOfMusicians;

    Ensemble(int size) { this.numberOfMusicians = size; }

    public int numberOfMusicians() { return numberOfMusicians; }

}
```

apa yang enum katakan tentang ordinal : sebagian besar programmer tidak akan mau menggunakan method ini. Ini di desain untuk digunakan oleh enum dengan tujuan yang lebih general untuk struktur data seperti EnumSet dan EnumMap. Kecuali kalau kita sedang coding struktur data, ordinal method adalah pilihan terbaik.

Item 32. Use EnumSet instead of (daripada) Bit Fields.

Enumerated element di dalam sebuah sets sudah normal menggunakan enum pattern integer.

```
// Bit field enumeration constants - OBSOLETE!

public class Text {

        public static final int STYLE_BOLD = 1 << 0; // 1

        public static final int STYLE_ITALIC = 1 << 1; // 2

        public static final int STYLE_UNDERLINE = 1 << 2; // 4

        public static final int STYLE_STRIKETHROUGH = 1 << 3; // 8


        // Parameter is bitwise OR of zero or more STYLE_ constants

        public void applyStyles(int styles) { ... }

}
```

kode di atas menganjurkan kamu menggunaka operasi bitwise OR untuk dikombinasikan dengan beberapa konstanta ke dalam Set, dikenal sebagai Bit Field.

```
text.applyStyles(STYLE_BOLD | STYLE_ITALIC);
```

Bit field juga menampilkan pada kita bahwa operasi bitwise aritmatika lebih efisien untuk melakukan operasi gabungan dan irisan. Tapi bit field punya semua kekurangan yang dimiliki konstanta enum integer dan masih banyak lagi. Bahkan lebih sulit untuk menggunakan bit field daripada konstanta enum integer jika di printed sebagai angka/bilangan. Juga, tidak gampang untuk melakukan iterasi pada semua nilai di dalam bit field, atau yang merupakan instance bit field.

Paket java.util menyediakan kelas EnumSet untuk lebih efisien dalam menyajikan kumpulan nilai yang diperoleh dari tipe enum tunggal. Kelas ini meng-implement Set interface, menyediakan semua kelebihan, aman, dan mudah dioperasikan dengan semua jenis Set implementatio lainnya. Tetapi di dalamnya, tiap-tiap Enum Set tersusun atas vektor bit. Enum Set jauh lebih

baik dan aman untuk performance dibandingkan dengan bit field. Banyak operasi yang sulit yang mesti dikerjakan manual dengan bit field, tetapi udah di bundle dalam Enum Set, sehingga menghindarkan kita dari kesalahan akibat mesti di coding manual, contohnya adalah Bulk Operation, seperti removeAll() dan retailAll().

Perhatikan di bawah ini jika kita mengganti code bit field di atas dengan menggunakan EnumSet. Terlihat lebih bersih dan aman codenya.

```
// EnumSet - a modern replacement for bit fields

public class Text {

        public enum Style { BOLD, ITALIC, UNDERLINE, STRIKETHROUGH }

        // Any Set could be passed in, but EnumSet is clearly best

        public void applyStyles(Set<Style> styles) { ... }

}
```

Lihat di bawah ini, client melempar sebuah instance dari EnumSet ke dalam applyStyles method. Enum Set menyediakan creator of set yang static dan sangat mudah digunakan, sederhana.

```
text.applyStyles(EnumSet.of(Style.BOLD, Style.ITALIC));
```

Melihat di atas, applyStyles method menggunakan parameter Set<Styles> dari pada EnumSet<Styles>. Sedikit terlihat seperti semua client akan melempar parameter berupa EnumSet ke dalam method, adalah pilihan yang baik untuk menerima tipe interface daripada implementation-nya. Ini menghandle kemungkinan jika saja client yang aneh melempar Set implementation yang berbeda dari EnumSet dan terlihat benar tidak ada kerugian.

Simpulannya, hanya karena sebuah tipe enumerated akan digunakan dalam Set, tidak ada alasan untuk menggunakan bit fields. Enum Set merupakan

kombinasi performance dari bit field ditambah dengan lebih banyak kelebihannya yang merupakan kelebihan tipe enum.

Hanya ada sebuah kekurangan yang nyata, EnumSet tidak memungkinkan untuk membuat immutable EnumSet.

Item 33. Use EnumMap instead of ordinal indexing.

Di awal, kita sudah pernah membahas mengenai ordinal method yang bisa membaca index dari konstanta tipe enum. Lihat di bawah sebuah kelas yang simple merepresentasikan herb.

```
public class Herb {

        public enum Type { ANNUAL, PERENNIAL, BIENNIAL }

        private final String name;

        private final Type type;

        Herb(String name, Type type) {

                this.name = name;

                this.type = type;

        }

        @Override public String toString() {

                return name;

        }

}
```

Sekarang saatnya membayangkan, seandainya kita ingin membuat sebuah array yang menggambarkan tanaman di kebun. Kita akan memisahkan mereka menjadi 3 tipe, yaitu annual, perennial, atau biennial. Untuk ini, biasanya programmer membuat 3 buah set, dan melakukan iterasi di seluruh kebun, menempatkan tiap-tiap herb di masing-masing Set. Sebagian

programmer akan melakukan ini, yaitu meletakkan semua sets ke dalam array berdasarkan index yang dari ordinal.

```
// Using ordinal() to index an array - DON'T DO THIS!

Herb[] garden = ... ;

// Indexed by Herb.Type.ordinal()

Set<Herb>[] herbsByType = (Set<Herb>[]) new Set[Herb.Type.values().length];

for (int i = 0; i < herbsByType.length; i++)

        herbsByType[i] = new HashSet<Herb>();


for (Herb h : garden)

        herbsByType[h.type.ordinal()].add(h);


// Print the results

for (int i = 0; i < herbsByType.length; i++) {

        System.out.printf("%s: %s%n",

        Herb.Type.values()[i], herbsByType[i]);

}
```

Teknik ini berjalan dengan baik, tetapi membawa banyak masalah. Karena array tidak compatible untuk Generics, aplikasi akan memerlukan banyak casting dan process compile tidak akan mulus. Karena array tidak tahu apa yang direpresentasikan oleh index, kita harus menuliskan label dari konstanta enum secara manual. Tapi, masalah yang paling serius adalah, ketika kita akan mengakses array yang indeksnya merupakan ordinal dari konstanta enum, akan menjadi tanggung jawab kita untuk menggunakan nilai integer yang tepat, sebab nilai integer tidak menjami tipe enum kita

safety. Jika kita menggunakan nilai integer yang salah, maka dia akan melakukan operasi yang salah, atau kalau beruntung kita akan mengalami runtime error, yaitu ArrayIndexOutOfBoundsException.

Untungnya masih ada cara yang lebih baik untuk menghasilkan efek yang sama. Array merupakan sebuah map yang secara efektif memapping antara enum dengan nilainya, sehingga sebaikknya kita bisa menggunakan Map. Lebih spesific lagi, ada implementasi dari Map yang dirancang khusus untuk digunakan buat enum type, namanya EnumMap. Perhatikan code di bawah ketika kita modify code di atas dengan EnumMap.

```
// Using an EnumMap to associate data with an enum

Map<Herb.Type, Set<Herb>> herbsByType = new EnumMap<Herb.Type, Set<Herb>>(Herb.Type.class);

for (Herb.Type t : Herb.Type.values())

        herbsByType.put(t, new HashSet<Herb>());

for (Herb h : garden)

        herbsByType.get(h.type).add(h);

System.out.println(herbsByType);
```

Code menjadi lebih ringkas, bersih dan aman dibandingan kecepatan dengan versi ordinal method. Tidak ada kemungkinan casting yang berpotensi membuat error. tidak perlu menulis manual label dari tiap-tiap konstantan yang akan di tampilkan, sebagaimana map kesy adalah enum yang tahu gimana mentranslate konstanta itu menjadi tulisan kembali, dan tidak mungkin ada error dalam komputasi index dari array. Alasan kenapa EnumMap lebih cepat dari versi original yang ordinal adalah perbandingan kecepatan di ordina index, karena EnumMap menggunakan array di dalamnya, tetap dirahasikan kepada programmer, kombinasi EnumMap Konstruktor akan memperkaya dan menigkatkan kecepatan array mereka.

perhatikan contoh di bawah ini yang akan membuat enum di dalam enum, sehinggan index dari enum memiliki index lagi, karena mempunyai enum lainnya sebagai nilainya, jadi code ini memapping enum Phase ke enum Transition, contoh di bawah ini adalah contoh yang dibuat dengan menggunakan ordinal.

```
// Using ordinal() to index array of arrays - DON'T DO THIS!

public enum Phase { SOLID, LIQUID, GAS;

    public enum Transition {

        MELT, FREEZE, BOIL, CONDENSE, SUBLIME, DEPOSIT;

        // Rows indexed by src-ordinal, cols by dst-ordinal

        private static final Transition[][] TRANSITIONS = { { null, MELT, SUBLIME },

                { FREEZE, null, BOIL },

                { DEPOSIT, CONDENSE, null }

        };

        // Returns the phase transition from one phase to another

        public static Transition from(Phase src, Phase dst) {

            return TRANSITIONS[src.ordinal()][dst.ordinal()];

        }

    }

}
```

lalu selanjutnya di bawah ini merupaka code yang menggunakan EnumMaps, hasil modifikasi dari code dengan ordinal di atas.

```
// Using a nested EnumMap to associate data with enum pairs
```

```
public enum Phase {
        SOLID, LIQUID, GAS;
        public enum Transition {
                MELT(SOLID, LIQUID), FREEZE(LIQUID, SOLID),
                BOIL(LIQUID, GAS), CONDENSE(GAS, LIQUID),
                SUBLIME(SOLID, GAS), DEPOSIT(GAS, SOLID);
                final Phase src;
                final Phase dst;
                Transition(Phase src, Phase dst) {
                        this.src = src;
                        this.dst = dst;
                }
                // Initialize the phase transition map
                private static final Map<Phase, Map<Phase,Transition>> m =
                        new EnumMap<Phase, Map<Phase,Transition>>(Phase.class);
                        static {
                                for (Phase p : Phase.values())
                                        m.put(p,new
                                EnumMap<Phase,Transition>(Phase.class));
                                for (Transition trans : Transition.values())
                                        m.get(trans.src).put(trans.dst, trans);
                        }
                        public static Transition from(Phase src, Phase dst) {
                                return m.get(src).get(dst);
```

```
        }
    }
}
```

Bandingkan kedua code di atas, bayangkah seandainya kita akan menambahkan kosntanta enum baru di code original yang menggunakan ordinals, akan jauh lebih sulit dan rawan untuk error, sedangkan yang kedua lebih sederhana, kita hanya langsung menambahkan saja, misalnya ***--*** *To update the EnumMap-based version, all you have to do is add PLASMA to the list of phases, and IONIZE(GAS, PLASMA) and DEIONIZE(PLASMA, GAS) to the list of phase transitions.* ***--***

Simpulannya adalah, hindarkan menggunakan ordinal method untuk enumerated enum, gunakanlah EnumMap, lebih efektif, bersih codenya, dan aman untuk jangkan panjang, mudah dilakukan perawatan alias maintenace.

Item 34. Emulate exensible enum with interface.

Cukup perhatikan contoh di bawah ini.

```
// Emulated extensible enum using an interface
public interface Operation {
    double apply(double x, double y);
}
public enum BasicOperation implements Operation {
    PLUS("+") {
        public double apply(double x, double y) { return x + y; }
    },
    MINUS("-") {
```

```
            public double apply(double x, double y) { return x - y; }
        },
        TIMES("*") {
            public double apply(double x, double y) { return x * y; }
        },
        DIVIDE("/") {
            public double apply(double x, double y) { return x / y; }
        };
        private final String symbol;
        BasicOperation(String symbol) {
            this.symbol = symbol;
        }
        @Override public String toString() {
            return symbol;
        }
    }
```

Satu Contoh Lagi.

```
// Emulated extension enum
public enum ExtendedOperation implements Operation {
    EXP("^") {
        public double apply(double x, double y) {
            return Math.pow(x, y);
        }
    },
```

```
        REMAINDER("%") {
            public double apply(double x, double y) {
                return x % y;
            }
        };
        private final String symbol;
            ExtendedOperation(String symbol) {
            this.symbol = symbol;
        }
        @Override public String toString() {
            return symbol;
        }
}
```

sedikit kekurangan dari menggunakan interface untuk emulate extensible enum adalah implementasi tidak bisa diwariskan dari satu enum type ke enum type lainnya. Dalam kasus di atas, logicnya adalah kita mengambil symbol dari operasi dan menyimpannya dengan operasi yang duplikasi antara di Basic Operation and ExtendedOperation. Dalam kasus ini, nggak masalah karena duplikasi kode hanya sedikit, tapi untuk berikutnya seandainya fungsionalitas yang akan di shared lebih banyak dan luas lagi, maka kita bisa membuat sebuah kelas pembungkus yang berupa helper kelas, atau static helper kelas untuk membuat duplikasi.

In summary, while you cannot write an extensible enum type, you can emulate it by writing an interface to go with a basic enum type that implements the interface. This allows clients to write their own enums that

implement the interface. These enums can then be used wherever the basic enum type can be used, assuming APIs are written in terms of the interface.

Item 35. Prefer Annotations to Naming Patterns.

Format pemberian nama adalah hal yang umum digunakan untuk memberikan label penjelasan. Yang pada umumnya diberikan secara manual, misalnya untuk melakukan testing dengan JUnit, kita memberikan nama dengan diawali kata test, misalnya testSafetyOverride. Cara ini bisa digunakan, tetapi memiliki banyak kekurangan.

1. salah ketik nama bisa menyebabkan error tanpa diketahui. Misalnya, mau menulis, testSafetyOverride, salah menjadi stetSafetyOverride. JUnit tidak akan mendeteksi itu sebagai error, tapi menyebabkan issue pada security.
2. Pola penamaan adalah bahwa tidak ada cara untuk memastikan bahwa mereka digunakan hanya pada elemen program yang sesuai. Misalnya, Anda memanggil testSafetyMechanisms kelas dengan harapan bahwa JUnit secara otomatis akan menguji semua metode, terlepas dari nama mereka. Sekali lagi, JUnit tidak akan complain, tetapi test tidak akan dijalankan dengan baik.
3. Tidak ada mekanisme yang valid untuk membuat relasi antara nilai parameter dengan element dari program. Misalnya, sebuah tes yang sukses adalah ketika kita mendapati sebuah exception tertentu. Exception sebenarnya adalah parameter dari tes tersebut. Bisa saja kita membuat sebuah nama tertentu untuk menyinbolkan sebuah exception tersebut yang akan digunakan dalam method tertentu yang akan digunakan dalam testing. Tapi itu akan menjadikannya buruk dan rapuh. Compiler tidak ada cara untuk mengetahui bahwa nama tersebut merupakan exception yang sebenarnya dijalankan. Jika nama kelas tidak ada or bukan sebuah exception, kita tidak akan tahu itu sebelum kita me-running code dan kita akan mendapati error, barulah kita tahu.

Annotation mempunyai solusi yang baik terhadap semua permasalahan itu. Andai saja kita ingin membuat annotations yang dirancang untuk melakukan

testing otomatis dan gagal jika mendapati sebuah exception tertentu. Silakan lihat contoh di bawah ini, annotaion dengan nama Test.

```
// Marker annotation type declaration

import java.lang.annotation.*;

/**

* Indicates that the annotated method is a test method.

* Use only on parameterless static methods.

*/

@Retention(RetentionPolicy.RUNTIME)

@Target(ElementType.METHOD)

public @interface Test {

}
```

Kelas Annotation di atas tersusun atas annotation Retention dan Target, kedua-nya dikenal sebagai meta-annotation. RententionPolicy.RUNTIME menyatakan bahwa kelas Test annotation akan dieksekusi ketika runtime. Tanpa itu, Kelas Test Annotation tidak akan muncul di jendela test tool. Target - ElementType.METHOD menyatakan bahwa Kelas Test Annotation hanya diijinkan untuk method yang di deklarasikan, tidak bisa diimplementasikan untuk kelas, tipe data/field atau element program yang lainnya.

Perhatikan komentar di atas deklarasi kelas Test Annotaion, “Hanya Berguna Untuk Method Static Tanpa Parameter”. Luar biasa jika saja complier dapat mengimplementasikan itu, tetapi nyatamya tidak bisa. Terdapat batasan seberapa banyak error checking yang dapat dilakukan oleh compiler, sekalipun itu dengan annotation. Jika kita letakkan kelas Test Annotaion di atas deklarasi sebuah non-static method atau method dengan satu atau lebih parameter, proses testing akan tetap dijalankan, membiarkan testing tool untuk menghandle issue ketika runtime.

Perhatikan potongan kode program di bawah ini, menjelaskan gimana Test annotaion bekerja.

```
// Program containing marker annotations

public class Sample {

@Test public static void m1() { } // Test should pass

public static void m2() { }

@Test public static void m3() { // Test Should fail

throw new RuntimeException("Boom");

}

public static void m4() { }

@Test public void m5() { } // INVALID USE: nonstatic method

public static void m6() { }

@Test public static void m7() { // Test should fail

throw new RuntimeException("Crash");

}

public static void m8() { }

}
```

kode di atas disebut marker annotation, karena tidak punya parameter tapi sederhananya langsung menandai elementnya. Jika programmer salah tulis Test, atau menerapkan untuk element dari program bukannya methodnya, program tidak akan bisa di compile.

Contoh di atas memiliki 8 static kelas, 4 menggunakan annotation test, dua diantaranya menjalankan exception, dan sisanya tidak. Tetapi salah satu dari mereka merupakan non-static method, sehingga itu bukan cara yang valid. Simpulannya, terdapat 4 tes, 1 berhasil, 2 gagal, dan 1 invalid. Sisa 4

method yang tidak menggunakan Test annotaion akan diabaikan oleh testing tool.

Test annotation tidak memiliki efek langsung pada semantic dari kelas Sample. Itu hanya menyediakan informasi untuk kepentingan program. Lebih luasnya, annotation tidak pernah mengubah semantic dari code yang menggunakan annotaion, tapi mengaktifkan special treatment oleh tools, seperti contoh di bawah ini.

```
//Program to process marker annotations

import java.lang.reflect.*;


public class RunTests {


    public static void main(String[] args) throws Exception {


        int tests = 0;

        int passed = 0;


        Class<?> testClass = Class.forName(args[0]);


        for (Method m : testClass.getDeclaredMethods()) {


            if (m.isAnnotationPresent(Test.class)) {


                tests++;
```

```
                    try {
                        m.invoke(null);
                        passed++;
                    } catch (InvocationTargetException wrappedExc) {
                        Throwable exc = wrappedExc.getCause();
                        System.out.println(m + " failed: " + exc);
                    } catch (Exception exc) {
                        System.out.println("INVALID @Test: " + m);
                    }
                }
            }
            System.out.printf("Passed: %d, Failed: %d%n", passed, tests - passed);
        }
}
```

Jika kita menjalankan code di atas dan menggunakan kelas Sample untuk di tes, maka akan mendapatkan, hasil seperti di bawah ini.

```
public static void Sample.m3() failed: RuntimeException: Boom
INVALID @Test: public void Sample.m5()
public static void Sample.m7() failed: RuntimeException: Crash
Passed: 1, Failed: 3
```

Sekarang, mari tambahkan code yang mendukung untuk tes yang sukses hanya jika mendapatkan exception tertentu.

```
//Annotation type with a parameter

import java.lang.annotation.*;

/**

* Indicates that the annotated method is a test method that

* must throw the designated exception to succeed.

*/

@Retention(RetentionPolicy.RUNTIME)

@Target(ElementType.METHOD)

public @interface ExceptionTest {

        Class<? extends Exception> value();

}
```

Maksud dari annotation di dalam ExceptionTest adalah kelas apaan yang meng-extends Exception, dan juga mengizinkan pada pengguna dari annotation untuk specify bermacam jenis exception. Lihat code di bawah untuk caranya.

```
public class SampleWithException {

        @ExceptionTest(ArithmeticException.class)

        public static void m1() { // Test should pass

                int i = 0;

                i = i / i;

        }

        @ExceptionTest(ArithmeticException.class)

        public static void m2() { // Should fail (wrong exception)

                int[] a = new int[0];
```

```
        int i = a[1];
    }
    @ExceptionTest(ArithmeticException.class)
    public static void m3() { } // Should fail (no exception)
}
```

Ubah code RunTests sebelumnya menjadi seperti di bawah ini.

```
public class RunTestWithException {
    public static void main(String[] args) throws Exception {
        int tests = 0;
        int passed = 0;
        Class<?> testClass = Class.forName(args[0]);
        for (Method m : testClass.getDeclaredMethods()) {
            if (m.isAnnotationPresent(ExceptionTest.class)) {
                tests++;
                try {
                    m.invoke(null);
                    System.out.printf("Test %s failed: no exception
%n", m);
                } catch (InvocationTargetException wrappedEx) {
                    Throwable exc = wrappedEx.getCause();
                    Class<? extends Exception> excType =

    m.getAnnotation(ExceptionTest.class).value();
                    if (excType.isInstance(exc)) {
```

```
                    passed++;
                } else {
                    System.out.printf("Test %s failed: expected
%s, got %s%n",
                            m, excType.getName(), exc);
                }
            } catch (Exception exc) {
                System.out.println("INVALID @Test: " + m);
            }
        }
        System.out.printf("Passed: %d, Failed: %d%n", passed, tests -
passed);
    }
}
```

Cara kerja dari code di atas mirip dengan RunTests sebelumnya, annotation akan mengecheck apakah ada exception yang di throw atau tidak pada sebuah kelas SampleWithException.

Seandainya kita menggunakan parameter annotaion berupa array of class object, maka kita memodifikasi code di atas menjadi seperti di bawah ini.

```
//Annotation type with an array parameter
import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface ExceptionTestArrayObject {
```

```
    Class<? extends Exception>[] value();
}
```

Kelas Sample dengan array of class object sebagai parameter.

```
import java.util.ArrayList;
import java.util.List;
public class SampleArrayObject {
    // Code containing an annotation with an array parameter
    @ExceptionTestArrayObject({ IndexOutOfBoundsException.class,
    NullPointerException.class })
    public static void doublyBad() {
        List<String> list = new ArrayList<String>();

        // The spec permits this method to throw either
        // IndexOutOfBoundsException or NullPointerException
        list.addAll(5, null);
    }
}
```

Tes yang dilakukan dengan parameter berupa array of class object.

```
//Program to process marker annotations
import java.lang.reflect.*;
public class RunTestArrayObject {

    public static void main(String[] args) throws Exception {
```

```
        int tests = 0;
        int passed = 0;
        Class<?> testClass = Class.forName(args[0]);
        for (Method m : testClass.getDeclaredMethods()) {
            if (m.isAnnotationPresent(ExceptionTestArrayObject.class)) {
                tests++;
                try {
                    m.invoke(null);
                    System.out.printf("Test %s failed: no exception
%n", m);
                } catch (Throwable wrappedExc) {
                    Throwable exc = wrappedExc.getCause();
                    Class<? extends Exception>[] excTypes =

    m.getAnnotation(ExceptionTestArrayObject.class).value();
                    int oldPassed = passed;
                    for (Class<? extends Exception> excType :
excTypes) {
                        if (excType.isInstance(exc)) {
                            passed++;
                            break;
                        }
                    }
                    if (passed == oldPassed)
```

```
                                        System.out.printf("Test %s failed: %s %n",
m, exc);
                                    }
                                }
                }
                System.out.printf("Passed: %d, Failed: %d%n", passed, tests -
passed);
        }
}
```

Testing framework yang kita buat sebagai contoh pada kode di atas lebih seperti sebuah mainan, tapi menjelaskan pada kita dengan gamblang kekuatan dari annotation ketimbang pola penamaan biasa, dan itu hanya sebagian kecil apa yang bisa kita lakukan dengan annotation. Jika kita meulis sebuah tool yang membutuhkan programmer untuk memberikan informasi tambahan ke sebuah file, definisikanlah sekumpulan annotation type yang tepat. Tidak ada alasan sederhana apapun untuk tetap menggunakan pola penamaan biasa, sedangkan kita telah mengenal annotation.

Item 36. Consistently Use The Override Annotation.

Ketika annotation di launching pada release java 1.5, beberapa annotation library baru telah ditambahkan, untuk beberapa typical programmer, yang paling penting diantara semua adalah Override. Annotation ini hanya dapat digunakan pada method, dan itu menandakan bahwa method yang meng-annotate override menimpa/meng-override Override si supertypenya. Jika menggunakannya secara konsisten, kita akan dilindungi dari bugs. Perhatikan kode program di bawah ini, cobalah untuk menemukan bugnya.

```
//Can you spot the bug?
public class Bigram {
```

```
    private final char first;
    private final char second;
    public Bigram(char first, char second) {
        this.first = first;
        this.second = second;
    }
    public boolean equals(Bigram b) {
        return b.first == first && b.second == second;
    }
    public int hashCode() {
        return 31 * first + second;
    }
    public static void main(String[] args) {
        Set<Bigram> s = new HashSet<Bigram>();
        for (int i = 0; i < 10; i++)
            for (char ch = 'a'; ch <= 'z'; ch++)
                s.add(new Bigram(ch, ch));

        System.out.println(s.size());
    }
}
```

Hasil dari kode program di atas adalah 260, padahal seharusnya adalah 26, kenapa itu bisa terjadi ? dimanakah letak kesalahannya ?

Jelas terlihat bahwa, programmer ingin melakukan override pada equals method, bahkan tidak lupa pula untuk meng-everride method hashcode. Sayangnya, programmer salah atau gagal meng-override method equlas, malahan melakukan overloading method equals. Untuk meng-override method equals, kita mesti melempar parameter dengan tipe object, tetapi parameter ynag dilempar adalah Bigram, bukan Object.

Untuk itu kita mesti modifikasi kode program equals, menjadi seperti di bawah ini.

```
@Override
public boolean equals(Object o) {
        if (!(o instanceof Bigram))
                return false;
        Bigram b = (Bigram) o;
        return b.first == first && b.second == second;
}
```

Oleh karena itu, kita mesti menggunakan override annotation untuk method yang kita inginkan untuk modifikasi fungsional dari method superclassnya.

In Summary, compiler dapat melindungi kita dari error yang parah jika kita menggunakan override annotation pada setiap method yang kita yakini untuk melakukan override pada superclassnya. Pada konkret kelas, kita tidak perlu untuk melakukan annotate override pada method yang dideklarasikan pada abstract kelas yang kita implement.

Item 37. Use Marker Interface to Define Types.

Marker Interface adalah sebuah interface yang tidak memiliki method, tapi merancang sebuah kelas yang meng-implementnya memiliki beberapa

property. Contohnya, pikirkan tentang Serializable interface. Dengan meng-implement itu, instance dari kelas dapat ditulis ke ObjectOuputStream.

Mungkin kita pernah mendengar bahwa marker annotation membuat marker interface tidak berguna, pengertian itu tidak benar, kelebihan dari marker interface yang pertama dan paling utama adalah, marker interface mendefinisikan sebuah type yang di-implementasikan oleh instance dari kelas yang di-marked tersebut, yang marker annotation tidak bisa lakukan. Kehadiran type ini mengizinkan kita untuk menemukan error pada saat compile, yang hanya bisa kita temukan ketika runtime untuk yang menggunakan marker annotation.

Kapan kita menggunakan marker interface dan marker annotation ? jelasnya, kita dapat menggunakan marker annotation jika marker diterapkan untuk program element yang spesifik, bukan untuk keseluruhan kelas atau interface.

Jika kita tidak ingin menambahkan method atau informasi apapun pada sebuah kelas, maka kita dapat menggunakan marker interface, jika sebaliknya maka gunakanlah marker annotation.

Marker Interface

"*An interface is called a marker interface when it is provided as a handle by Java interpreter to mark a class so that it can provide special behaviour to it at runtime and they do not have any method declarations****".***

Java Marker Interface Examples

- java.lang.Cloneable
- java.io.Serializable
- java.util.EventListener

Some examples of Java marker interfaces are :-

1) Serializable

2) Cloneable

3) RandomAccess

4) SingleThreadModel

5) EventListner

Chapter 07. Method

Item 38. Check Parameter for Validity.

Terdapat conventions yang umum pada parameter yang akan dilempar untuk method tertentu, sperti misalnya tidak boleh nilai index dari array berupa nilai negatid atau null. Kita mesti seceepat mungkin untuk mengetahui nilai yang dikirim itu valid atau tidak, karena akan menyebabkan issue yang sulit dimengerti seandainya kita gagal mendeteksinya sesaat setelah program masuk fase runtime.

Untuk method-emthod yang bersifat public, kita bisa menggunakan @throws, yang telah disediakan oleh java sendiri, seperti misalnya IllegalArgumentException, IndexOutBoundsException atau NullPointerException. Jikalai kita telah menambahkan itu, akan lebih mudah untuk memastikan jenis issue yang kita dapatkan pada saat program dieksekusi. Silakan lihat contoh code program di bawah ini.

```
/**
     * Returns a BigInteger whose value is (this mod m). This method
     * differs from the remainder method in that it always returns a
     * non-negative BigInteger.
     *
     * @param m the modulus, which must be positive
```

```
     * @return this mod m
     * @throws ArithmeticException if m is less than or equal to 0
     */
     public BigInteger mod(BigInteger m) {
          if (m.signum() <= 0)
               throw new ArithmeticException("Modulus <= 0: " + m);
          // Do the computation
          return m;
     }
```

Untuk non-public method kita bisa menggunakan assertation. Perhatikan contoh kode program dibawah ini.

```
// Private helper function for a recursive sort
     @SuppressWarnings("unused")
     private static void sort(long a[], int offset, int length) {
          assert a != null;
          assert offset >= 0 && offset <= a.length;
          assert length >= 0 && length <= a.length - offset;
          // Do the computation
     }
```

Simpulannya, tiap kali kita membuat sebuah method atau konstruktor, kita seharusnya berpikir tentang, apa yang tidak boleh terjadi pada semua parameter. Kita seharusnya menemukan semua yang tidak boleh itu dan melakukan pengecekan di tiap-tiap awal method body. Sangat penting untuk menjadikan ini sebuah kebiasaan. Pekerjaan ini akan dibayar dengan lunas

jika saja terjadi kegagalan diawal, saat pengecekan validitas dari method gagal.

Item 39. Make Defensive Copies When Needed.

Satu hal yang membuat java sangat baik adalah bahasa pemrograman yang aman, tidak seperti C dan C++. Tapi bahkan di yang aman pun, kita mesti berhati-hati pada setiap bagian yang rawan. Kita harus membuat kode program yang benar-benar aman, dengan asumsi bahwa client yang menggunakan program kita akan melakukan cara terbaik untuk merusak kode program yang aman itu. Itu bisa benar-benar terjadi jika seseorang ternyata coba untuk menembus dan merusak sistem keamanan program kita. Perhatikan kode program di bawah ini.

```
//Broken "immutable" time period class

public final class Period {


    private final Date start;


    private final Date end;


    /**
    * @param start the beginning of the period
    * @param end the end of the period; must not precede start
      * @throws IllegalArgumentException if start is after end
    * @throws NullPointerException if start or end is null
    */
```

```
    public Period(Date start, Date end) {
        if (start.compareTo(end) > 0)
            throw new IllegalArgumentException(


        start + " after " + end);
        this.start = start;
        this.end = end;
    }
    public Date start() {
        return start;
    }
    public Date end() {
        return end;
    }
    // Remainder omitted
}
```

awalnya, kelas ini bersifat immutable, dan untuk memaksa ketidak konsistenannya bahwa start in period tidak mengikuti end-nya.

```
// Attack the internals of a Period instance
Date start = new Date();
Date end = new Date();
Period p = new Period(start, end);
end.setYear(78); // Modifies internals of p!
```

Untuk melindungi object Period dari ketidak-konsistenan yang sengaja dilakukan, adalah penting untuk membuat sebuah copy nilai tiap-tiap mutable parameter ke constructor.

```
// Repaired constructor - makes defensive copies of parameters
    public Period(Date start, Date end) {

        this.start = new Date(start.getTime());
        this.end = new Date(end.getTime());
        if (this.start.compareTo(this.end) > 0)
            throw new IllegalArgumentException(start +" after "+ end);
    }
```

Catatan, defensive copies dibuat sebelum melakukan pngecekan validitas untuk paarameter, dan pengecekan validitas dilakukan untuk copiannya bukan aslinya.

Second attack of internal, perhatikan kode di bawah ini.

```
    public void secondAttack(){
        // Second attack on the internals of a Period instance
        Date start = new Date();
        Date end = new Date();
        Period p = new Period(start, end);
        p.end().setYear(78); // Modifies internals of p!
    }
```

modifikasi accessor menjadi seperti di bawah ini.

```
// Repaired accessors - make defensive copies of internal fields
```

```
public Date start() {
        return new Date(start.getTime());
}
public Date end() {
        return new Date(end.getTime());
}
```

Simpulannya, jika sebuah kelas memiliki mutable komponen yang didapat dari atau di return ke client, kelas harus melakukan defensive copy terhadap semua komponennya. Jika harga untuk melakukan itu tidak sebanding, dan kelas percaya bahwa client tidak akan melakukan modifikasi terhadap komponen dengan tidak bertanggung jawab, kemudian defensive copy dapat diganti dengan aggreement bahwa tanggung jawab client adalah tidak memodifikasi komponen yang akan berdampak besar jika dilakukan.

Item 40. Design Method Signatures Carefully.

1. Gunakan nama method yang benar, sesuai dengan standar. Nama method harus mudah dimengerti dan konsisten dengan nama lainnya di dalam paket yang sama.
2. Jangan berlebihan dalam menyediakan method yang sesuai. Setiap method seharusnya berbobot, fungsional dan tepat. Terlalu banyak method membuat kelas menjadi sulit untuk dipelajari, digunakan, didokuemntasikan, di-test dan di-maintain.
3. Hindari daftar parameter yang terlalu banyak atau panjang. Buatlah jumlah parameter maksimal 4 ataua kurang dari itu, kebanyakan programmer sulit untuk mengingat daftar parameter yang terlalu banyak. Parameter yang berurut terlalu panjang dan diketik sangat berbahaya. Bukan hanya user tidak mampu untuk mengingat urutan yang benar dari parameter, tapi jika urutan salah, program akan tetap di-compile dan running, tanpa pembuat sadar bahwa program telah salah.

Terdapat 3 teknik untuk mempersingkat daftar parameter yang panjang,

1. Pecah sebuah method yang kompleks menjadi method yang kecil-kecil, tiap-tiap dari method itu memuat sejumlah parameter tertentu. Kelemahannya, method akan terkesan begitu banyak, tapi itu bisa juga membantu untuk mengurangi jumlah dari method.
2. Membuat kelas helper yang menampung kumpulan parameter. Kelas helper merupakan anggota dari static kelas. Cara ini dianjurkan jika kita secara berkala dan konsisten menggunakan parameter yang mencerminkan entity yang tunggal dan spesifik. Contohnya, kita membuat sebuah kelas yang merepresentasikan card game, kita mengetahui bahwa kita secara konsisten melempar dua parameter secara berurutan yang merepresentasikan ranking dari kartu dan tipenya. Akan sangat membantu jika kita menambahkan sebuah kelas helper yang merepresentasikan card game dan menggantikan tiap kemunculan 2 parameter yang berurutan dengan parameter tunggal dari helper kelas tersebut.
3. Mengkombinasikan dua hal diatas, dengan mengadaptasi Builder Pattern dari pembentukan object ke pemanggilan method. Jika kita memiliki method dengan banyak parameter, khususnya jika sebagian merupakan parameter yang optional, itu bisa memungkinkan untuk membuat sebuah objek yang menggambarkan semua parameter itu, dan mengizinkan client membuat banyak setter di object itu, tiap-tiap dari single parameter, kelompok terkait. Seketika setelah paramater di setting, client mengeksekusi method execute di object itu, yang melakukan pengecekan terakhir untuk parameter dan melakukan komputasi yang teranyar.

1. Untuk tipe parameter, lebih baik interface daripada kelas. Karena dengan interface sebagai tipe-nya, kita bisa dengan leluasa untuk passing semua object yang meng-implement interface itu, sedangkan dengan kelas, kita harus passing tipe object yang spesifik.

2. Lebih baik enum dua element daripada parameter tipe boolean. Membuat kode program kita menjadi lebih muda dibaca dan ditulis, khususnya jika kita

sedang enggunakan IDE yang mendukung fitur autocomplete. Lebih mudah pula untuk menambahkan pilihan baru kedepannya.

```
public enum TemperatureScale { FAHRENHEIT, CELSIUS }
```

Perhatikan kode di atas, bukan hanya sederhana, tapi juga fleksibel seandainya kita mau menambahkan Kelvin ke dalamnya.

Item 41. Gunakan Overloading dengan Bijaksana.

Perhatikan kode di bawah ini.

```
// Broken! - What does this program print?
public class CollectionClassifier {
        public static String classify(Set<?> s) {
                return "Set";
        }
        public static String classify(List<?> lst) {
                return "List";
        }
        public static String classify(Collection<?> c) {
                return "Unknown Collection";
        }
        public static void main(String[] args) {
                Collection<?>[] collections = {
                new HashSet<String>(),
                new ArrayList<BigInteger>(),
                new HashMap<String, String>().values()
        };
```

```
        for (Collection<?> c : collections)
            System.out.println(classify(c));
        }
}
```

Kita akan berharap bahwa program akan melakukan printing Set, List dan Unknown Collection ke layar monitor. Tapi nyatanya tidak, program akan melakukan printing Unknown Collection ke layar sebanyak 3 kali, kenapa itu bisa terjadi, karena overloading. Classify method overloading. Pilihan overloading mana yang akan di eksekusi diputuskan ketika compiling program berlangsung. Untuk 3 kali iteras yang diakukan, tipe parameter saat compile time adalah sama, yaitu Collection<?>. Tipe runtime berbeda tiap kali iterasi, tetapi itu tidak mempengaruhi pilihan dari overloading. Karena sekali lagi tipe saat compile time adalah Collection<?>.

Pilihan diantara semua overloading method bersifat static, sedangkan diantar overridden method bersifat dynamic. Versi yang benar dari override method diputuskan ketika runtime, berdasarkan tipe object saat runtime yang menentukan method mana yang akan di eksekusi.

Perhatikan kode program dibawah ini, sebagai pembanding dengan overloading diatas.

```
class Wine {
        String name() { return "wine"; }
}
class SparklingWine extends Wine {
        @Override
        String name() { return "sparkling wine"; }
}
class Champagne extends SparklingWine {
```

```
    @Override
    String name() { return "champagne"; }
}
public class Overriding {
    public static void main(String[] args) {
        Wine[] wines = { new Wine(), new SparklingWine(), new Champagne()};
        for (Wine wine : wines)
            System.out.println(wine.name());
    }
}
```

Untuk kode program di atas, hasil yang kita dapatkan berbeda dengan overloading tadi, kita mendapatkan ketiga hasil sesuai dengan object yang terdapat dalam Array of Wine tersebut.

Item 42. Gunakan Varargs dengan Bijaksana.

Varargs Method dikenal juga dengan sebutan variable arity method. Varargs method menerima nol atau lebih parameter dengan tipe yang spesifik. Varargs dipakai dengan cara menempatkan parameter-parameter di dalam sebuah array dan array inilah yang akan menjadi parameter dari method.

Perhatikan kode program di bawah ini.

```
// Simple use of varargs
    static int sum(int... args) {
        int sum = 0;
        for (int arg : args)
            sum += arg;
```

```
        return sum;
    }
```

Umumnya ketika kita akan melempar parameter melalui method, kita sudah tahu berapa banyak parameter yang akan dilempar, tetapi seandainya kita belum tahu kita bisa melakukannya dengan Varargs ini. Jumlah parameter yang dilepmar akan dianggap sebagai array of parameter atau argument.

```
System.out.println(SimpleVarargs.sum(1,2,3));

System.out.println(SimpleVarargs.sum());
```

Dua baris kode program di atas akan memberikan nilai yang berbeda, yaitu 6 dan 0.

Seandainya kita ingin melakukan sorting untuk mencari nilai minimum dari kumpulan parameter yang dlempar ke method tertentu, dan kalau kita menggunakan cara diatas, dengan hanya varargs parameter yang dilempar, itu akan sangat buruk dan rapuh. Karena kita butuh untuk mengecek terlebih dahulu, apakah varargs yang dilempar memiliki nilai atau zero. Perhatikan kode program dibawah ini.

```
// The WRONG way to use varargs to pass one or more arguments!
    static int min(int... args) {
        if (args.length == 0)
            throw new IllegalArgumentException("Too few arguments");
        int min = args[0];
        for (int i = 1; i < args.length; i++)
            if (args[i] < min)
                min = args[i];
        return min;
    }
```

Untungnya kita memiliki cara lain yang lebih baik. Perhatikan kode program di bawah ini.

```
// The right way to use varargs to pass one or more arguments
    static int min(int firstArg, int... remainingArgs) {
        int min = firstArg;
        for (int arg : remainingArgs)
            if (arg < min)
                min = arg;
        return min;
    }
```

Varargs dirancang sebenernya untuk menampilkan nilai ke layar (printf). Lihat contoh di bawah cara menampilkan nilai dari array.

```
// The right way to print an array
System.out.println(Arrays.toString(myArray));
```

Perhatikan kode program di bawah ini, untuk menjadikan kumpulan parameter menjadi sebuah List tertentu.

```
public static <T> List<T> gather(T... args) {
    return Arrays.asList(args);
}
```

Method varargs juga bisa di overload, contohny seperti di bawah ini.

```
tampilIsi(int ..args)
tampilIsi(boolean ..args)
tampilIsi(String name, int ...args)
```

Kelemahannya, setelah di overload, jangan membuat parameternya menjadi kosong, atau tanpa parameter, karena akan menyebabkan error. Antara tipe parameter varargs dan parameter yang lainnya tidak boleh sama.

Item 43. Return Empty Array or Collection, not nulls.

Terkadang ada argument yang menyatakan bahwa, lebih baik return null daripada array kosong karena menyangkut dengan alokasi memori. Pernyataan ini salah dalam dua hal.

1. Tidak bisa diterima untuk mengkhawatirkan performa pada level ini, kecuali method ini benar2 memiliki bagian dalam pengaruh terhadap issue tentang performance.
2. Memungkinkan untuk return array kosong untuk setiap pemanggilan yang me-return array kosong yang immutable, dan object yang immutable bebas untuk di sharing.

Simpulannya, tidak ada alasan yang tepat untuk me-return array atau collection apapun sebagai null. Mereka harus me-return array atau collection kosong.

Item 44. Write doc comments for all exposed API elements.

Tuliskan dokumentasi mengenai apapun dari koe program kita, dengan memberi comments diatas-nya. Untuk memberikan dokument pada API kita, kita harus mendahului tiap-tiap komponen dengan komentar, untuk semuanya, kelas, interface, konstruktor, method, dan field declaration atau variable. Kita harus menjelaskan dengan detail mengenai ciri-ciri dan fungsional dari mereka.

Dokumentasi berupa komentar untuk setiap method harus dituliskan dengan ringkas sesuai dengan kesekapatan kontrak dengan client. Komentar harus menjelaskan tentang apa dari emthod itu, bukan bagaimana. Harus mengelaborasi tentang precondition dari method, apa yang harus benar jikalau client akan mengeksekusi itu, dan pracondition-nya. Detail tentang

method itu, setelah method success dieksekusi apa lagi, terus mengenai exception dan throws jika ada. Intinya kita harus menjelaskan detail mengenai seluruh item yang menjadi bagian dari method itu dan menjelaskan pula flow atau alur process yang dilakukan di method itu. Dokumentasi berupa komentar juga harus menjelaskan mengenai thead safety dari sebuah kelas atau method yang bersangkutan. Untuk contohnya, perhatikan ilustrasi untuk method di bawah ini.

```
/**
        * Returns the element at the specified position in this list.
        *
        * <p>This method is <i>not</i> guaranteed to run in constant
        * time. In some implementations it may run in time proportional
        * to the element position.
        *
        * @param index index of element to return; must be
        * non-negative and less than the size of this list
         * @return
        * @return the element at the specified position in this list
        * @throws IndexOutOfBoundsException if the index is out of range
        * ({@code index < 0 || index >= this.size()})
        */
        public <T> Object get(int index) {return null;}
```

Ilustrasi Konstruktor.

```
/**
* A college degree, such as B.S., {@literal M.S.} or Ph.D.
```

```
* College is a fountain of knowledge where many go to drink.
*/
public class Degree {}
```

Ilustrasi Interface

```
/**
* An object that maps keys to values. A map cannot contain
* duplicate keys; each key can map to at most one value.
*
* (Remainder omitted)
*
* @param <K> the type of keys maintained by this map
* @param <V> the type of mapped values
*/
public interface Map<K, V> {}
```

Ilustrasi Enum Type.

```
/**
* An instrument section of a symphony orchestra.
*/
public enum OrchestraSection {
        /** Woodwinds, such as flute, clarinet, and oboe. */
        WOODWIND,
        /** Brass instruments, such as french horn and trumpet. */
        BRASS,
        /** Percussion instruments, such as timpani and cymbals */
```

```
        PERCUSSION,

        /** Stringed instruments, such as violin and cello. */

        STRING;

}
```

Ilustrasi Annotation Type.

```
/**

* Indicates that the annotated method is a test method that

* must throw the designated exception to succeed.

*/

@Retention(RetentionPolicy.RUNTIME)

@Target(ElementType.METHOD)

public @interface ExceptionTest {

/**

* The exception that the annotated test method must throw

* in order to pass. (The test is permitted to throw any

* subtype of the type described by this class object.)

*/

        Class<? extends Exception> value();

}
```

Chapter 8.

General Programming.

Item 45. Minimize the scope of local variable.

1. Deklarasikan variabel dimana dia dibutuhkan. Jika kita mendeklarasikan variabel sebelum dia dibutuhlan, itu akan mmpersulit untuk menerka alur

dari method. Mendeklarasikan method secara prematur akan menyebabkan scope dari variabel tidak hanya dimulai terlalu awal, tapi diakhiri terlalu lama.

2. Harusnya setiap local variabel yang dideklarasikan langsung di-initialize. Jika kita tidak mengetahui nilai untuk inisialiasasi local variabel, maka seharusnya kita menunda untuk medeklarasikan itu. Looping hadir dengan cara istimewa dalam membatasi scope dari local variabel, karena dengan mendeklarasikan local variabel di dalam sebuah looping, kita telah memberikan identitas kepada local variabel, bahawa di-butuhkan hanya di dalam looping itu.
3. Jadikan method itu kecil dan focus. Jangan menggabungkan dua aktivitas ke dalam satu method, karena local variabel bisa saja memiliki relevansi antara dua aktivitas itu.

Item 46. Prefer for-each loops to traditional for loops.

Looping dengan for lebih baik daripada looping dengan while statement, tapi itu bukan cara terbaik. Iterator dan index muncul atau ditulis tiga kali, dan itu memungkinkan untuk terjadi error. Masalahnya adalah, jika error terjadi maka for belum tentu bisa menghandle-nya.

The preferred idiom untuk looping adalah for-each, perhatikan contoh code di bawah ini.

```
// The preferred idiom for iterating over collections and arrays
for (Element e : elements) {
        doSomething(e);
}
```

Masalah akan muncul ketika kita memiliki nested for, dan seringkali terjadi.

```
// Can you spot the bug?
enum Suit { CLUB, DIAMOND, HEART, SPADE }
```

```
enum Rank { ACE, DEUCE, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN,
JACK, QUEEN, KING }

Collection<Suit> suits = Arrays.asList(Suit.values());

Collection<Rank> ranks = Arrays.asList(Rank.values());

List<Card> deck = new ArrayList<Card>();

for (Iterator<Suit> i = suits.iterator(); i.hasNext(); )
        for (Iterator<Rank> j = ranks.iterator(); j.hasNext(); )
                deck.add(new Card(i.next(), j.next()));
```

Contoh berikutnya,

```
// Same bug, different symptom!

enum Face { ONE, TWO, THREE, FOUR, FIVE, SIX }

Collection<Face> faces = Arrays.asList(Face.values());

for (Iterator<Face> i = faces.iterator(); i.hasNext(); )
        for (Iterator<Face> j = faces.iterator(); j.hasNext(); )
                System.out.println(i.next() + " " + j.next());
```

Contoh lainnya,

```
// Fixed, but ugly - you can do better!

for (Iterator<Suit> i = suits.iterator(); i.hasNext(); ) {
        Suit suit = i.next();
        for (Iterator<Rank> j = ranks.iterator(); j.hasNext(); )
                deck.add(new Card(suit, j.next()));
}
```

Solusi dengan for-each.

```
// Preferred idiom for nested iteration on collections and arrays
```

```
for (Suit suit : suits)
    for (Rank rank : ranks)
        deck.add(new Card(suit, rank));
```

Bukan hanya dengan for each, kita juga bisa menggunakan sebuah interface Iterable, yang hanya memiliki satu method yang berfungsi sama dengan for-each statement.

```
public interface Iterable<E> {
    // Returns an iterator over the elements in this iterable
    Iterator<E> iterator();
}
```

Simpulannya, for-each looping menyediakan kelebihan yang banyak ketimbang for looping yang traditional, dalam hal kejelasan dan pencegahan bug, tanpa efek dari sisi perform. Sayangnya, ada 3 kondisi dimana for-each looping tidak bisa digunakan.

1. Filtering-jika kita butuh melewati tiap-tiap element dan menghapus element yang tertentu.
2. Transforming-jika kita butuh untuk memodifikasi isi dari array atau list dengan menggunakan index.
3. Parallel Iteration-jika kita inigin melintasi banyak koleksi secara parallel, ingin pula mengontrol process iteration atau index dari variabel.

Jika kita berada pada salah satu dari kondisi ini, pastikan untuk menggunalan for looping yang traditional, hati-hati dengan jebakan yang telah disebutkan dibahasan ini, pastikan kita sedang melakukan yang terbaik yang bisa.

Item 47. Know and use libraries.

Bayangkan kita akan melakukan generating niali integer secara random antar 0-angka tertentu, programmer umumnya akan menulis kode seperti di bawah ini.

```
private static final Random rnd = new Random();

// Common but deeply flawed!

static int random(int n) {

        return Math.abs(rnd.nextInt()) % n;

}
```

kode di atas benar dan terlihat tepat, tapi masih terdapat banyak kekurangan secara teknis matematis, kita harus benar-benar mengetahui cara kerja random tersebut, bagaimana kalau seandainya nilai n terlalu kecil, sama atau terlalu besar, apa nilai yang akan dihasilkan, kita perlu melakukan checking pada nilai n untuk menyesuaikancara kita meng-handle kerja atau alur dari method, dan itu merepotkan. Tapi tenang saja, java telah menyediakan cara yang tepat, yaitu :

Random.nextInt(int)

kita tidak usah pusing-pusing lagi memikirkan cara kerjanya, karena syntax itu telah di-uji oleh para pakar dan melewati tahapan pengujian yang benar.

Dengan menggunakan library yang standar, kita mendapatkan keuntungan atas pengetahuan para expert yang menulis kode itu dan pengalaman semua orang yang pernah menggunakan ini sebelumnya.

Keuntungan kedua, kita tidak mesti menghabiskan waktu untuk menulis solusi tambahan yang berhubungan dengan permasalahan yang kita hadapi. Kita lebih baik konsentrasi pada aplikasi yang sedang dibangun ketimbang hal-hal seperti itu.

Keuntungan ketiga, performa dari aplikasi akan meningkat, tanpa kita yang mengerjakannya. Karena sebagian besar orang menggunakan hal itu dan itu merupakan standar yang telah diakui dalam dunia industri, organisasi yang mengembangkannya memiliki alasan yang mendasar untuk membuat itu lebih cepat dari yang lain.

Keuntungan terakhir, kita membuat code yang sesuai dengan standar koding yang baik. Kode program kita mudah di baca, mudah di-maintain, dan reusable oleh banyak developer lainnya.

Item 48. Hindari float dan double jika jawaban pasti yang dibutuhkan.

Jangan gunakan float dan double untuk perhitungan yang memerlukan jawaban yang presisi. Gunakan BigDecimal jika kita ingin sistem mencatat nilai decimal dan kita tidak memikirkan ketidaknyamanan dan harga untuk tidak menggunakan tipe data primitif. Menggukanan BigDecimal telah memberikan keuntungan untuk mengontrol kemampuan pembulatan. Gunakan int atau long jika bilangannya tidak terlalu besar, tidak lebih dari 9 digit gunakan int, tidak lebih dari 18 digit gunakan long, jika lebih dari 18 digit gunakan DigBecimal.

Item 49. Prefer primitive types to boxed primitives

Java terdiri dari 2 tipe, yaitu tipe data primitif, antara lain int, double, dan boolean, dan tipe data reference, antara lain String dan List. Setiap data primitif memiliki keterkaitan dengan tipe data reference, disebut boxed primitif. Boxed primitif untuk int, double dan boolean adalah Integer, Double, dan Boolean.

Terdapat 3 perbedaan yang mendasar diantara keduaya,

1. tipe data primitif hanya membawa nilai saja, sedangkan boxed primitif memiliki identitas yang berbeda dari nilai-nilai mereka. Dengan kata lain, dua boxed primitif bisa memiliki nilai yang sama, tapi identitas yang berbeda.
2. tipe data primitif hanya memiliki nilai fungsional tunggal, sedangkan tiap-tiap boxed primitif punya nilai non-functional tunggal lagi, yaitu null, disamping semua nilai-nilai tipe data primitif yang bersesuaian.
3. tipe data primitif umumnya lebih efisien dalam waktu dan ruang memori daripada boxed primitif.

tiga perbedaan mendasar di-atas dapat memberi kita masalah yang besar jika tidak berhati-hati.

Perhatikan kode program dibawah ketika akan mengurutkan nilai dari besar ke kecil.

```
// Broken comparator - can you spot the flaw?
Comparator<Integer> naturalOrder = new Comparator<Integer>() {
        public int compare(Integer first, Integer second) {
                return first < second ? -1 : (first == second ? 0 : 1);
        }
};
```

secara kasat mata, Comparator terlihat baik dan standar. Dan akan sukses jika dilakukan banyak testing. Seandainya kita membandingkan nilai yang sama untuk first dan second, kita akan mendapatkan return value berupa 0, tapi kita akan mendapatkan 1, kenapa itu terjadi, karena kita membandingkan alamat reference dari dua value yang berbeda, sehingga nilai yang kita dapat adalah 1. Itu kenapa menggunakan == untuk melakukan perbandingan nilai boxed primitif akan selalu salah. Kita seharusnya menggunakan seperti kode program dibawah ini.

```
Comparator<Integer> naturalOrder = new Comparator<Integer>() {
        public int compare(Integer first, Integer second) {
                int f = first; // Auto-unboxing
                int s = second; // Auto-unboxing
                return f < s ? -1 : (f == s ? 0 : 1); // No unboxing
        }
};
```

selanjutnya kode program di bawah ini,

```
public class Unbelievable {

        static Integer i;

        public static void main(String[] args) {

        if (i == 42)

                System.out.println("Unbelievable");

        }

}
```

apakah hasil yang di dapat jika nilai i di-inisiliasasi dengan 42, tulisan Unbelieveable, tidak, anda salah, kita akan mendapatkan NullPointerException, masalahnya adalah i adalah Integer dan bukan int, itu merefleksikan reference dari nilai 42, dan kita membandingkan Integer dengan int, dan hampir di setiap kasus, jika kita menggunakan primitf dan boxed primitif dengan operasi yang sama maka variabel boxed primitif akan mengalami auto-unboxed. Jika null object reference mengalami auto-unboxed, maka kita akan mendapatkan hasil seperti di-atas.

Akhirnya, kapan kita seharusnya menggunakan boxed primitif, mereka memiliki beberapa kondisi untuk digunakan.

1. sebagai element, key, atau nilai di dalam koleksi, Kita tidak bisa menggunakan tipe data primitif dalam koleksi, sehingga kita dipaksa untuk menggunakan boxed primitif.
2. Ini merupakan kasus khusus, kita mesti menggunakan boxed primitif di dalam parameterixed type, karena java tidak mengizinkan untuk menggunakan tipa data primitif. Contohnys, kita tidak bisa men-declare ThreadLocal<int>, jadi harus menggunakan ThreadLocal<Integer>.
3. Akhirnya, kita harus menggunakan boxed primitif ketika membuat reflective method invocation.

Item 50. Avoid String where other type are more approriate

String dirancang untuk me-representasikan teks, dan mereka melakukan itu dengan sangat baik. Karena String sangat umum dan sangat didukung oleh java, ada kecenderungan menggunakan String untuk sesuatu yang menyalahi dari untuk apa mereka diciptakan. Inilah rincian hal-hal yang tidak seharusnya dilakukan dengan String.

1. String adalah pengganti yang buruk untuk nilai dari tipe data lain. Ketika memperoleh data dari file, network atau inputan dari keyboard, mereka lebih banyak dalam bentuk String. Itu adalah kecenderungan umum untuk itu, tapi itu dibenarkan jika datanya benar-benar merupakan teks, jika datanya numeric, kita harus meng-convert nilainya menjadi int, float atau BigInteger. Jika berupa yes-no answer, harus di translate ke nilai boolean. Lebih umum lagi, jika ada value type yang sesuai, entah itu primitif atau reference type, kita harus meng-convertnya.
2. String adalah pengganti yang buruk untuk tipe data enum. Enum lebih baik bertipe konstan daripada String.
3. String adalah pengganti yang buruk untuk tipe aggregasi. Lihat contoh dibawah ini,

```
// Inappropriate use of string as aggregate type

String compoundKey = className + "#" + i.next();
```

Lebih baik membuat sebuah kelas tersendiri untuk merepresentasikan aggregasi seperti di-atas.

4. String adalah pengganti yang buruk untuk capabilitas. Lihat kode program di bawah ini.

```
// Broken - inappropriate use of string as capability!

public class ThreadLocal {

    private ThreadLocal() { } // Noninstantiable



    // Sets the current thread's value for the named variable.
```

```
    public static void set(String key, Object value);


    // Returns the current thread's value for the named variable.
    public static Object get(String key);
}
```

Masalah dari kode program di-atas adalah, String key merupakan global namespace yang di-share untuk client, sedangkan String key harus unique, bagaimana seandainya client memiliki key yang sama, dan akan mengakibatkan 2 client gagal akses pada saat yang bersamaan, dan keamanan memburuk, antar client bisa saja mengakses data yang bukan haknya karena kunci akses-nya sama. Maka dari itu untuk mengatasi ini, silakan lihat kode program di-bawah ini.

```
public class ThreadLocal {
    private ThreadLocal() { } // Noninstantiable
    public static class Key { // (Capability)
        Key() { }
    }
// Generates a unique, unforgeable key
    public static Key getKey() {
        return new Key();
    }
    public static void set(Key key, Object value);
    public static Object get(Key key);
}
```

Simpulannya, hindari kecenderungan umum untuk menggunakan String jika terdapat tipe data yang lebih tepat untuk digunakan. Penggunaan yang tidak tepat, String adalah lebih tidak praktis, tidak fleksibel, lambat, dan rawan kesalahan daripada tipe data lain.

Item 51. Beware the performance of string concatenation.

Operator concatenation dari String (+) adalah cara paling nyaman untuk mengkombinasikan kumpulan string kecil menjadi satu kesatuan String besar. Tidak masalah untuk output yang berupa single line yang singkat atau untuk menciptakan String yang kecil, objek dengan size spesifik, tapi itu tidak terukur. Menggunakan operator konkatenasi dari String untuk memotong atau menggabungkan n strings membutuhkan waktu senilai quadrat dari n. Itu adalah konsekuensi karena String bersifat immutable, ketika dua buah String akan di konkatenasi, maka keduanya akan di-copy.

```
// Inappropriate use of string concatenation - Performs horribly!
public String statement() {
    String result = "";
    for (int i = 0; i < numItems(); i++)
        result += lineForItem(i); // String concatenation
        return result;
}
```

Untuk meningkatkan performance gunakan StringBuilder menggantikan String.

```
public String statement() {
    StringBuilder b = new StringBuilder(numItems() * LINE_WIDTH);
    for (int i = 0; i < numItems(); i++)
        b.append(lineForItem(i));
```

```
            return b.toString();
    }
}
```

Konsepnya sederhana, jangan menggunakan operator konkatenasi String, atau performa yang dihasilkan akan buruk. Gunakan StringBuilder's append method.

Item 52. Refer to object by their interface.

Item 40 menjelaskan, kita seharusnya menggunakan interface sebagai tipe data dari parameter. Lebih luas lagi, kita harus menggunakan interface daripada kelas untuk mereferensikan objek. Jika tipe interface yang bersesuaian ada, kemudian paramater, nilai return, variabel, dan field seharusnya dideklarasikan menggunakan tipe interface. Waktu satu-satunya dimana kita harus menggunakan kelas untuk merefer-objek adalah ketika kita membuat objek menggunakan konstruktor. Untuk membuat ini jadi konkret, pertimbangkan Vector yang meng-implement interface dari list. Biasakan untuk menggunakan seperti di bawah ini.

```
// Good - uses interface as type
List<Subscriber> subscribers = new Vector<Subscriber>();
```

rather than this:

```
// Bad - uses class as type!
Vector<Subscriber> subscribers = new Vector<Subscriber>();
```

Jika kita telah membiasakan diri untuk menggunakan interface, program kita akan lebih fleksibel. Jika kita memutuskan untuk mengganti implementasi, yang harus kita lakukan hanyalah mengganti nama kelas yang menjadi konstruktor untuk menggunakan static factory yang berbeda.

```
List<Subscriber> subscribers = new ArrayList<Subscriber>();
```

Melakukan penggantian nama kelas adalah jika itu menawarkan lebih banyak kentungan, seperti lebih cepat dan hemat memori.

Lebih baik me-refer objek dengan kelas daripada interface jika tidak ada tipe interface yang sesuai. Contohnya,

1. Bayangkan value classes, antara lain String dan BigInteger. Value classes umumnya ditulis dengan banyak implementasi di dalam pikiran. Mereka seringnya final dan umumnya memiliki interface yang berhubungan. Sempurna untuk menggunakan value classes sebagai parameter, variabel, field atau nilai return. Lebih Luas, jika sebuah konkret kelas tidak dihubnngkan dengan interface, kemudian kita tidak punya pilihan untuk me-refer-nya, pilihan acak jatuh pada kelas-nya sendiri.
2. Jika sebuah object termasuk framework kelas base, itu lebih bagus untuk me-refer itu ke kelas base yang relevant, yang mana secara khusus abstrak, daripada implementasi dari kelasnya. Java.util.TimerTask cock dengan kategori ini.

Item 53. Prefer interface to reflection.

Core reflection facility, java.lang.reflect, menawarkan akses untuk informasi tentang kelas. Dengan menggunakan nama dari kelas tertentu, kita bisa memdapatkan informasi tentang konstruktor, method dan field yang dideklarasikan.

Item 54. Use Native method secara bijaksana

JNI(Java NAtive Interface) mengizinkan aplikasi java untuk memanggil native method, yang merupakan special method yang ditulis dalam bahasa pemrograman native seperti C atau C++. Native method bisa menampilkan komputasi yang sepihak dalam bahasa native/asli sebelum kembali menggunakan bahasa pemrograman java.

Native/asli method memiliki tiga kegunaan utama.

1. Mereka menyiapkan akses ke tempat-tempat yang spesific ke arah platform, seperti register computer dan file locks.
2. Mereka menyiapkan akses ke kode-kode rahasia, yang mana bisa kembali menyiapkan akses ke data-data rahasia.
3. Akhirnya, native/asli method digunakan untuk menuliskan bagian kritis performa dari aplikasi dalam bahasa native untuk meningkatkan performa.

Menggunakan native method untuk mengakses fasilitas yang terkait spesifik platform, tapi sebagaimana java platform yang telah lebih matang, itu menyediakan lebih dan lebih fitur yang sebelumnya ditemukan hanya di host platform. Sebagai contoh, java.util.prefs, ditambahkan dalam release java 4, menawarkan fungsionalitas dari register, dan java.awt.SystemTray, ditambahkan dalam release java 6, menawarkan akses ke area sistem tray dekstop. Itu juga cara yang sah untuk menggunakan native method untuk mengakses kode-kode penting.

Adalah cara yang tidak dianjurkan menggunakan native method untuk meningkatkan performa, tiap-tiap release VM selalu meningkatkan kinerja mereka dari versi yang lebih awal, kinerjanya pasti selalu lebih baik dan cepat. Contohnya, ketika java.math menambahkan BigInteger untuk meningkatkan operasi aritmatika yang multipresisi.

Menggunakan native method sebenarnya memiliki kekurangan yang fatal. Karena bahasa native tidaklah aman, aplikasi yang menggunakan bahasa native tidak jauh lebih kebal terhadap memori corruption error. Karena bahasa native adalah platform yang tidak bebas, aplikasi menggunakan native method jauh lebih tidak portable. Aplikasi menggunakan native code jauh lebih sulit untuk di-debug. Ada harga yang pasti harus dibayar ketika masuk dan keluar dari kode native, sehingga native method dapat menurunkan performa jika mereka hanya melakukan jumlah pekerjaan yang sedikit. Akhirnya, native method membutuhkan “glue code” yang sulit untuk dibaca dan tidak menarik untuk ditulis.

Simpulannya, pikirkan berkali-kali ketika akan menggunakan native method. Jarang, jika pernah, gunakan mereka untuk meningkatkan performa. Jika harus menggunakan native method untuk mengakses sumber daya yang low level atau kode-kode tertentu, gunakan sesedikit mungkin kode native jika mungkin dan lakukan testing berulang. Sebuah bug yang kecil saja bisa merusak keseluruhan aplikasi kita.

Item 55. Optimize Judiciously.

Ada tiga peribahasa terkait optimasi yang setiap orang seharusnya mengetahui.

1. Lebih berdosa dalam komptuasi adalah terlalu percaya dan memuja yang namanya efisiensi, dari pada alasan lainnya - termasuk orang buta yang bodoh.
2. Kita seharusnya melupakan tentang efisiensi yang kecil, katakn sekitar 97% dari waktu, optimasi yang prematur adalah akar dari semua kejelekan.
3. Kita mengikuti 2 aturan dalam optimasi.

    1. Jangan Lakukan itu.

    2. Jangan lakukan itu sebelumnya, tidak sampai kita memiliki sebuah solusi yang optimal dan jelas

Tiga peribahasa di-atas hadir dalam bahasa pemrograman java sekitar 2 decade lalu. Mereka mnegisahkan kebenaran yang dalam tentang optimasi. Lebih mudah untuk melakukan keburukan yang berlebihan daripada kebaikan, khususnya jika kita melakukan optimasi yang prematur. Pada prosesnya, kita mungkin membuat sebuah perangkat lunak tidak cepat atau tepat, dan tidak bisa dibereskan dengan gampang.

Jangan mengorbankan prinsip-prinsip perancangan perangkat lunak demi untuk performa. Berjuanglah untuk menulis program yang baik daripada cepat. Jika program yang baik tidak bisa jalan dengan cepat, maka rancangan yang baik akan meng-optimasi itu. Program yang baik mewujudkan prinsip menyembunyikan informasi : yang mana

memungkinkan, mereka melokalisasi keputusan design dalam modul tunggal, sehinggan keputusan tunggal bisa berubah tanpa menimbulkan efek pada keseluruhan program.

Ini tidak berarti kita bisa mengeyampingkan performa sampai program kita selesai. Masalah implementasi bisa dibereskan oleh optimasi yang dilakukan kemudian, tapi kelemahan design yang terkandung yang membatasi performa tidak memungkinkan untuk diselesaikan tanpa menulis ulang program. Mengubah bagian fundamental dari rancangan setelah melihat fakta bisa berakibat dalam rusaknya struktue dari sistem yang itu menyulitkan untuk perawatan dan pengembangan aplikasi. Oleh karena itu, kita mesti memikirkan tentang performa ketika sedang dalam proses perancangan.

Upayakan untuk menghindari rancangan yang membatasi peforma. Bagian yang paling sulit untuk dimodifikasi setelah terjadi adalah menentukan interaksi antara modul dengan dunia diluarnya. Yang paling utama diantar semua design component adalah API, wire-level protocol, dan format dari data yang disimpan. Bukan hanya semua data itu sulit atau tidak mungkin untuk dimodifikasi setelah terjadi, tapi semua dari mereka dapat menimbulkan hambatan yang significant pada apa yang sebenernya sistem dapat lakukan.

Pertimbangkan konsekuensi terhadap kinerja dari keputusan rancangan API yang telah di pilih. Membuat variabel bersifat public yang mutable memungkinkan kebutuhan banyak defensive copying yang tidak diperlukan. Sama saja dengan menggunakan pewarisan untuk kelas public dimana composition akan lebih tepat untuk digunakan untuk membuat hubungan selamanya dengan superclassnya, yang mana bisa menimbulkan keterbatasan dalam kinerja dari subclassnya.

Simpulan, jangan upayakan menulis program yang cepat, upayakan menulis program yang baik, dan kecepatan akan datang. Jangan berpikir tentang

kinerja ketika kita sedang merancang sistem dan khususnya ketika sedang merancang API, wire-level protocols, dan format data yang disimpan. Ketika program tuntas dibuat, barulah tes kinerjanya. Jika telah cukup cepat, maka kerja kita beres. Jika tidak, temukan masalah pada sumber dayanya dengan bantuan profiler, dan lakukan optimasi pada bagian yang relevan dengan sistem tersebut.

1. Periksa pilihan algoritma kita. tidak banyak optimasi low level bisa menimbulkan pilihan algoritma yang buruk. Ulangi process ini, ukur kinerja setelah setiap perubahan sampai kita merasa puas.

Item 56. Adhere to generally accepted naming conventions.

Platform java telah meyediakan kumpulan cara penamaan yang sesuai aturan, kebanyak dari mereka telah terdaftar di Java Language Spesification (JLS). Langsung saja, standar penamaan dibagi menjadi 2 kategori, Penulisan dan Tata Bahasa.

1. Penulisan.

Identifier Type	Examples
Package	`com.google.inject, org.joda.time.format`
Class or Interface	`Timer, FutureTask, LinkedHashMap, HttpServlet`
Method or Field	`remove, ensureCapacity, getCrc`
Constant Field	`MIN_VALUE, NEGATIVE_INFINITY`
Local Variable	`i, xref, houseNumber`
Type Parameter	`T, E, K, V, X, T1, T2`

2. Grammatical.

Package : No Grammatical Convention

Classes, including enum Type : terdiri dari satu atau dua kata benda, Timer, BufferedWriter

Interface : seperti Class, Collection, Comparator, atau kata sifat dengan akhiran able atau ible, seperti Runnable, Iterable, atau Accessible.

Annotation : Katab benda, kata kerja, kata depan, dan kata sifat, BindingAnnotation, Inject, ImplementedBy, atau Singleton.

Method : kata kerja atau frase kata kerja, append dan drawImage.

Method return boolean : diawali dengan is atau has dan diikuti oleh kata benda, frase kata benda, kata apapun, frase yang berfungsi sebagai kata sifat, isDigit, isProbablePrime, isEmpty, isEnabled, atau hasSiblings

Method return non-boolean : kata benda, frase kata benda, frase kata kerja diawali oleh get, size, hascode, atau getTime.

Fields : kata benda dan frase kata benda, height, digits, atau bodyStyle

Fields boolean type : di-awali dengan is,

Chapter 9. Exception

Item 57. Use Exception only for exceptional conditions.

Terkadang, jika kita tidak beruntung, kita akan menemukan potongan kode program seperti di bawah ini.

```
// Horrible abuse of exceptions. Don't ever do this!
try {
        int i = 0;
        while(true)
                range[i++].climb();
} catch(ArrayIndexOutOfBoundsException e) {
```

```
}
```

Apa yang terjadi pada kode program diatas, adalah akan terjadinya looping selamanya, tetapi akan mendapatkan exception ArrayIndexOutOfBoundsException, karena array range tidak memiliki nilai atau bernilai null. Kode program di atas tidak dibenarkan, karena kita memaksakan exception akan terjadi seandainya range pada index tertentu bernilai null. Padahal kita bisa menggunakan kode program yang lebih general, yang programmer java sudah mengetahui persis.

```
for (Mountain m : range)
        m.climb();
```

nilai dari kasus ini adalah exception adalah, sebagaimana nama mereka menerangkan, hanya untuk digunakan bagi kondisi yang istimewa, mereka seharusnya tidak pernah digunakan untuk alur program yang umum.

Prinsip yang berkaitan langsung dengan design API adalah design API yang baik tidak memaksakan client untuk menggunakan exception untuk alur program yang umum.

Seperti contoh object Iterator di-bawah ini, mereka memiliki mekanisme sendiri dalam mengetahui kondisi dari objeknya. Iterator memiliki next yang digunakan untuk mengambil nilai berikutnya dari objek ini. Iterator juga memiliki method hasNext yang digunakan untuk mengetahui, apakah masih ada nilai pada index berikutnya di dalam objek Iterator, seperti kode di-bawah ini.

```
for (Iterator<Foo> i = collection.iterator(); i.hasNext(); ) {
        Foo foo = i.next();
}
```

Jika Iterator tidak memiliki method hasNext, maka client terpaksa harus menulis kode program seperti ini.

```
// Do not use this hideous code for iteration over a collection!
try {
        Iterator<Foo> i = collection.iterator();
        while(true) {
                Foo foo = i.next();
        }
} catch (NoSuchElementException e) {
}
```

Mirip seperti kode program di-awal, dan ini merupakan cara yang buruk untuk testing, karena bisa terjadi dugaan yang salah terhadap error apa yang terjadi, dan menyembunyikan error yang sebenarnya.

Simpulannya, exception dirancang untuk digunakan dalam kondisi yang istimewa, Jangan menggunakan mereka untuk alur program yang wajar atau umum, dan jangan menulis API yang memaksa yang lain melakukan itu.

Item 58. Use checked exceptions for recoverable conditions and runtime exception for programming errors.

Java menyediakan 3 jenis throwables.

1. checked exception
2. runtime exception
3. errors

Prinsip untuk menentukan apakah menggunakan checked exception atau un-checked exception adalah gunakan checked exception untuk kondisi yang mana user memungkinkan untuk bisa mengalami recovery. Dengan menggunakan checked exception, kita diharapkan bisa meng-handle exception dan melakukan recover dalam catch clause. Penggunaan un-checked exception dipilih seandainya jika kita menggunakan checked

exception akibat yang akan didapatkan adalah lebih buruk daripada menggunakan un-checked exception.

Tapi biasanya kita tidak selalu menemukan kasus yang hitam putih begitu, misalnya saja sebuah contoh kasus di bawah ini. Kita memiliki sebuah kasus memori yang overload, yang bisa disebabkan oleh peng-alokasian memori untuk array yang terlalu besar, jika kasus itu disebabkan oleh memori yang terlalu besar hanya sementara, maka kita bisa menggunakan checked exception untuk melakukan recovery. Jika yakin bahwa itu hanya sementara, maka kita bisa menggunakan checked exception, jika tidak, maka lebih baik menggunakan un-checked exception, runtime exception dan error.

Cara kerja checked exception, misalnya muncul error ketika melakukan pembayaran online, tapi gagal karena total uang tidak cukup, maka catch clause harus menyediakan akses untuk menambahkan jumlah uang di account itu sehingga proses pembayaran bisa berlangsung normal kembali.

Item 59. Avoid unnecessary use of checked Exception

Checked exception adalah fitur yang luar bisa dari pemrograman java. Tidak seperti kode return, checked exception memaksa programmer untuk peduli terhadap kondisi yang istimewa, pengembangan yang luar biasa handal. Penggunaan checked exception yang berlebihan bisa membuat sebuah API menjadi tidak menyenangkan untuk digunakan. Jika sebuah method mengalami satu atau lebih exception, maka kita mesti meng-handle itu dengan satu atau lebih blok catch juga, atau memastikan itu mengeksekusi sebuah statement throws dan membuat mereka gagal.

```
} catch(TheCheckedException e) {

        throw new AssertionError(); // Can't happen!

}
```

How about this?

```
} catch(TheCheckedException e) {
```

```
        e.printStackTrace(); // Oh well, we lose.
        System.exit(1);
}
```

Jika programmer yang menggunakan API tidak bisa melakukan yang lebih baik, sebuah un-checked exception akan menjadi lebih tepat. Contoh terkait exception yang gagal melewati test adalah CloneNotSupportedException. Yang dilemparkan oleh Object.clone, yang harus dipanggil hanya pada objek yang mengimplementasikan Cloneable. Faktanya, catch blok hampir selalu memiliki karakter character of an assertion failure. Checked exception secara natural tidak memberikan keuntungan untuk programmer, tapi membutuhkan usaha dan program yang kompleks.

Tambahan yang menyulitkan programmer, adalah seandainya, sebuah method memiliki try-catch blok, maka di kode program yang memanggil method itu harus memiliki try-catch blok juga.

Dibawah ini, checked exception yang dikonversi menjadi unchecked exception.

```
// Invocation with checked exception
try {
        obj.action(args);
} catch(TheCheckedException e) {
        // Handle exceptional condition
}
```

to this:

```
// Invocation with state-testing method and unchecked exception
if (obj.actionPermitted(args)) {
        obj.action(args);
```

```
} else {
        // Handle exceptional condition
}
```

Item 60. Favor the use of standard exceptions

Gunakanlah exception yang standar dan umum serta bisa di-reuse dalam implementasinya, karena itu akan membantu kode program yang kita tulis menjadi mudah dipelajari dan dimengerti, serta mudah untuk dibaca, juga terkesan lebih familiar dengan kebanyakn programmer. Terdapat banyak un-checked exception yang sudah dikenal dan secara umum terstandar.

Exception	Occasion for Use
`IllegalArgumentException`	Non-null parameter value is inappropriate
`IllegalStateException`	Object state is inappropriate for method invocation
`NullPointerException`	Parameter value is null where prohibited
`IndexOutOfBoundsException`	Index parameter value is out of range
`ConcurrentModificationException`	Concurrent modification of an object has been detected where it is prohibited
`UnsupportedOperationException`	Object does not support method

Item 61. Throw Exceptions approriate to the abstraction

Pastikan exception yang diterapkan pada lower level juga bekerja dengan baik pada higher layer level, karena jika terjadi ketidaksesuaian, maka program harus mengalami modifikasi yang cukup melelahkan.

Bahkan jika semua ini kejadian tanpa kita ketahui, maka kita bisa menyelesaikannya dengan yang disebut exception translation. Perhatikan contoh di-bawah ini yang merupakan standar exception untuk interface List<E>.

```
/**
* Returns the element at the specified position in this list.
* @throws IndexOutOfBoundsException if the index is out of range
* ({@code index < 0 || index >= size()}).
*/
public E get(int index) {
        ListIterator<E> i = listIterator(index);
        try {
                return i.next();
        } catch(NoSuchElementException e) {
                throw new IndexOutOfBoundsException("Index: " + index);
        }
}
```

Item 62. Document all exception thrown by each method

Sangat penting untuk mendokumentasikan semua exception di tiap-tiap method.

1. Selalu deklarasikan checked exception secara sendiri, dan cata secara tepat kondisi yang mana, tiap-tiap thrown menggunakan javadoc @throws tag.
2. Gunakan javadoc @throws tag untuk mendokumentasikan tiap-tiap unchecked exception yang d mungkin di throw di method tertentu, tapi jangan gunakan kata kunci throws untuk memasukkan unchecked esception di dalam deklarasi method.
3. Jika sebuah exception di-alami oleh banyak method di dalam kelas untuk alasan yang sama, sangat bisa diterima untuk mendokumentasikan exception di dalam dokumentasi kelas berupan komentar.

Simpulannya, catat semua exception yang di-alami semua method yang kita tulis.

Item 63. Include Failure-Capture information in detail messages

Untuk menangkap kegagalan, pesan yang detail dari sebuah exception seharusnya mengandung semua nilai dari parameter dan field yang memiliki andil/kontribusi dalan exception itu.

For example, instead of a String constructor, IndexOutOfBoundsException could have had a constructor that looks like this:

```
/**
* Construct an IndexOutOfBoundsException.
*
* @param lowerBound the lowest legal index value.
* @param upperBound the highest legal index value plus one.
* @param index the actual index value.
*/
public IndexOutOfBoundsException(int lowerBound, int upperBound, int index) {
        // Generate a detail message that captures the failure
        super(      "Lower bound: " + lowerBound +
                ", Upper bound: " + upperBound +
                ", Index: " + index);
        / Save failure information for programmatic access
        this.lowerBound = lowerBound;
        this.upperBound = upperBound;
        this.index = index;
}
```

Item 64. Strive for failure atomicity

Sesaat setelah terjadi exception dan di-throws, umumnya objek ada dalam kondisi yang baik, bahkan jika error muncul di tengah-tengah jalannya sebuah operasi tertentu. Ini khususnya terjadi pada kondisi checked exception, dikarenakan user ingin melakukan recovery terhadap proses yang gagal. Bahasa awamnya, gagalnya pemanggilan atau eksekusi method tertentu seharusnya meninggalkan objek pada kondisi yang siap untuk eksekusi berikutnya. Method yang mengalami kejadian seperti ini disebut sebagai failure atomic (kegagalan kecil).

Terdapat beberapa cara untuk mendapatkan atau mencapai kondisi ini. Cara terbaik adalah mendesain sebuah objek yang immutable. Jika sebuah objek bersifat immutable, maka failure atomicity is free. Jika sebuah operation gagal, itu akan mencegah objek baru diciptakan, tapi tidak membiarkan objek dalam kondisi yang tidak stabil atau tidak konsisten, karena kondisi objek akan konsisten jika dia diciptakan dan tidak dapat di modifikasi nilainya.

Untuk method yang menggunakan mutable objek, cara yang paling ternama untuk mencapai kondisi failure atomicity adalah dengan melakukan pengecekan ketika sebelum memproses parameter itu dalam sebuah operasi tertentu. Ini menyebabkan setiap exception akan mengalami throws sebelum objek mengalami perubahan.

perhatikan contoh kode program di bawah ini.

```
public Object pop() {
    if (size == 0)
        throw new EmptyStackException();
    Object result = elements[--size];
    elements[size] = null; // Eliminate obsolete reference
```

```
        return result;
}
```

Cara ketiga dan tidak terlalu umum adalah dengan membuat sebuah recovery code untuk melakukan intercept seandainya terjadi exception di tengah-tengah proses eksekusi dan mengembalikan kondisi objek pada kondisi semula seperti sebelum terjadi proses eksekusi.

Cara terakhir adalah membuat sebuah copy objek yang akan dieksekusi dalam sebuah method itu, dan segera menggantikan nilai dalam objek itu dengan copy-annya yang sudah disiapkan sebelumnya, dan setelah proses eksekusi complete barulah replace lagi dengan objek yang original.

Meskipun failure atomicity umumnya sangat berguna, tapi tidak selalu direalisasikan. Sebagai contoh, seandainya terdapat dua buah thread yang akan memanipulasi nilai pada sebuah objek yang sama secara simultan tanpa sinkronisasi, maka itu akan membuat objek dalam keadaan yang tidak konsisten. Akan menjadi salah jika kita beranggapan bahwa objek tersebut masih berguna setelah mengalami CurrentModificationException. Sebagaimana aturan mainnya, error adalah sesuatu yang tidak bisa ditanggulangi lagi, atau tidak bisa di-recovery, dan sebuah method tidak perlu bahkan untuk memelihara kondisi failure atomicity saat error di-throws.

Bahkan ktika failure atomocity memungkinkan, itu tidak selalu dibutuhkan. Untuk beberapa kasus, itu malah akan meningkatkan cost dan kompleksitas. Itu mengatakan, adalah mudah dan bebas untuk mencapai failure atomicity asal kita paham apa issue yang dihadapi.

Item 65. Don’t ignore exceptions.

Exception berguna untuk mendeteksi sebuah error atau kesalahan yang terjadi ketika program di-eksekusi, sehingga kita bisa mengetahui kenapa dan dimana potensi error itu meskipun tidak secara spesifik tahu jikalau banyak baris program dalam blok try-catch. Mengabaikan exception sama

saja membuang sebuah alarm kebakaran, dan ketika kebakaran terjadi kita tidak menyadari, hingga akhirnya terdapat banyak kerusakana di sana-sini. Jangan abaikan exception baik untuk checked exception maupun unchecked exception.

CHAPTER 10.

Concurrency.

Threads mengizinkan banyak aksi berjalan secara bersamaan dan simultan. Concurrent programming lebih sulit daripada single-thread programming, karena banyak hal yang bisa memburuk dan sangat sulit untuk melakukan reproduce. Tapi, kita tidak bisa menghindari konkurensi, karena banyak terjadi dalam dunia nyata, untuk menciptakan aplikasi yang powerful kita memerlukan fasilitas ini.

Item 66. Synchronize access to shared mutable data.

Keyword **synchronized** memastikan hanya satu thread yang bisa dieksekusi untuk method atau proses tertentu dalam satu waktu. Seorang programmer selalu takut sebuah inkonsisten objek di akses, tetapi keyword di atas memastikan bahwa sebuah objek hanya akan diakses ketika telah konsisten, dalam artian, proses yang mencoba mengakses itu akan berada dalam antrian untuk mengakses secara bergantian ketika proses yang pertama atau duluan sudah selesai, sehingga objek sudah dalam keadaan konsisten ketika di-akses.

Pendapat ini benar, tapi ini baru setengah dari cerita mengenai sinkronisasi. Tanpa sinkronisasi, perubahan yang dilakukan oleh satu thread tidak mungkin terdeteksi oleh thread berikutnya. Bukan hanya sinkronisasi mencegah akses saat objek inkonsisten, tapi memastikan tiap-tiap thread masuk ke proses sinkronisasi dan mendapatkan update tentang apa aja yang telah terjadi pada objek sebelumnya, sehingga objek berada pada kondisi

terakhir sebelum mengalami manipulasi atau diakses lagi dan langsung di-lock.

Language juga menjamin menulis dan membaca variable adalah atomic, kecuali kalau variabel bertipe long atau double. Dengan kata lain, membaca variabel selain long dan double digaransi untuk mengembalikan nilai yang disimpan dalam variabel oleh beberapa thread, bahkan jika banyak thread memanipulasi variabel secara konkuren dan tanpa sinkronisasi.

Sederhananya, tipe data selain long dan double akan mengembalikam nilai yang terbaru dari hasil manipulasi oleh satu thread atau banyak thread tanpa perlu sinkronisasi, ini didukung oleh java.

Kita mungkin pernah mendengar bahwa untuk meningkatkan kinerja, kita seharusnya menghindari sinkronisasi ketika membaca atau menulis atomic data. Saran ini berbahayanya adalah saran yang salah.

Sinkronisasi dibutuhkan untuk komunikasi yang tersedia antara semua thread, yang mana merumuskan tentang kapan dan bagaimana perubahan yang dilakukan oleh sebuah thread visible untuk thread yang lainnya dan mutual exclusion.

Mutual Exclusion adalah suatu cara yang menjamin jika ada sebuah proses yang menggunakan variabel atau berkas yang sama (digunakan juga oleh proses lain), maka proses lain akan dikeluarkan dari pekerjaan yang sama. Jadi, Mutual Exclusive terjadi ketika hanya ada satu proses yang boleh memakai sumber daya, dan proses lain yang ingin memakai sumber daya tersebut harus menunggu hingga sumber daya tadi dilepaskan atau tidak ada proses yang memakai sumber daya tersebut

Konsekuensi dari gagalnya proses sinkronisasi akses ke mutable data bisa menjadi mengerikan jika data adalah berupa atomic data yang dibaca dan ditulis. Pertimbangkan urusan untuk memberhentikan sebuah thread

dari thread lainnya. Java menyediakan command Thread.stop, tapi method ini sudah deprecated karena tidak aman/safe, penggunaannya bisa menyebabkan data menjadi corrupt. Sehingga, jangan sekali-kali menggunakan Thread.stop. Cara terbaik yang dapat digunakan adalah membuat thread pertama setting sebuah boolean variabel dengan nilai initial false, dan akan bisa diubah menjadi true sebagai tanda bahwa thread pertama dimatikan oleh thread kedua. Karena membaca dan menulis sebuah nilai boolean adalah atomic, karena atomic, itu menjadi visible untuk semua thread dan melakukan sinkronisasi secara otomatis.

```
// Broken! - How long would you expect this program to run?
public class StopThread {
        private static boolean stopRequested;
        public static void main(String[] args)
                throws InterruptedException {
                Thread backgroundThread = new Thread(new Runnable() {
                        public void run() {
                                int i = 0;
                                while (!stopRequested)
                                i++;
                        }
                });
                backgroundThread.start();
                TimeUnit.SECONDS.sleep(1);
                stopRequested = true;
        }
}
```

Cara kode di-atas bekerja adalah salah, cara terbaik untuk membenarkannya adalah melakukan sinkronisasi akses ke variabel stopRequested.

```
//Properly synchronized cooperative thread termination
public class StopThreadCorrect {
        private static boolean stopRequested;
        private static synchronized void requestStop() {
                stopRequested = true;
        }
        private static synchronized boolean stopRequested() {
                return stopRequested;
        }
        public static void main(String[] args) throws InterruptedException {
                Thread backgroundThread = new Thread(new Runnable() {
                        public void run() {
                                @SuppressWarnings("unused")
                                int i = 0;
                                while (!stopRequested())
                                        i++;
                        }
                });
                backgroundThread.start();
                TimeUnit.SECONDS.sleep(1);
                requestStop();
        }
```

}

catatan penting bahwa method write dan method read untuk variabel stopRequested keduanya di-sinkronisasi. Tidaklah cukup untuk hanya sinkronisasi method write saja. Faktanya, sinkronisasi tidak memberikan efek apapun kecuali kalau keduanya method read dan write disinkronisasi.

aksi sinkronisasi di dalam method stopThread akan atomic meskipun tanpa sinkronisasi. Dengan kata lain, sinkronisasi di dalam method semata-mata hanya berguna untuk komunikasi perubahan nilai, bukan untuk mutual exclusion. Ketika cost untuk sinkronisasi dalam tiap-tiap iterasi masih murah, ada jalan lain yang lebih tidak bertele-tele dan memiliki kinerja lebih baik. Proses locking di dalam stopThread bisa dihilangkan jika stopRequested dideklarasikan volatile. Ketika volatile modifier tidak untuk mutual exclusion, itu menjamin bahwa tiap thread yang membaca field akan melihat tiap-tiap nilai yang mengalami perubahan dalam proses penulisan value yang baru.

Namun tetap harus berhati-hati menggunakan volatile, volatile akan sangat berbahaya jika dideklarasikan untuk sebuah nilai yang mengalami perubahan terlalu sering, misalnya increment, kita ingin mengenerate sebuah serial number berdasarkan increment yang dipunya, suatu saat bisa saja sebuah thread salah membaca nilai increment tersebut, sehingga menyebabkan serial number hasil generate itu memiliki nilai yang sama dengan serial number sebelumnya, sehingga akan mengalami kegagalan, karena proses increment adalah proses read and write, mungkin saja thread membaca nilainya saat belum di-write, sehingga akan mengambil nilai sebelumnya, padahal nilai itu sudah dipakai sebelumnya.

Ada cara yang lebih baik yang telah disediakan oleh java untuk generate nilai, dalam library : java.util.concurrent.atomic, kita tidak perlu melakukan sinkronisasi lagi karena sudah punya kinerja lebih baik tanpa sinkronisasi.

private static final AtomicLong nextSerialNum = new AtomicLong();

public static long generateSerialNumber() {

```
        return nextSerialNum.getAndIncrement();
}
```

Cara terbaik untuk masalah yang kita bahas ini adalah, jangan share mutable data. Hanya share immutable data, atau jangan share sama sekali. Jika kita melakukan itu, catatlah itu buat dokumentasinya, karena akan sangat membantu dalam proses maintain dan pengembangan aplikasi berikutnya. Mungkin mereka akan menggunakan threads padahal kamu tidak peduli dengan itu sebelumnya.

Simpulannya, ketika multiple threads sharing mutable data, tiap-tiap thread yang membaca atau menuliskan nilai baru harus melakukan sinkronisasi. Tanpa sinkronisasi, tidak ada yang menjamin bahwa perubahan yang dilakukan oleh sebuah thread bisa dibaca oleh thread lainnya. Akibat dari gagalnya sinkronisasi mutable data yang di-sharing adalah liveness dan safety failure. Mereka itu adalah salah dua diantara semua failure yang sangat sukar di-debug. Mereka bisa kadang muncul kadang tidak, dan munculnya tergantung waktu tertentu, dan pola dari aplikasi akan berbeda-beda dari run yang pertama ke run yang berikutnya. Jika kita hanya membutuhkan komunikasi antar thread, bukan mutual exclusion, deklarasikan dengan modifier volatile dapat menjadi pilihan sebagai bagian dari sinkronisasi, tapi terkadang untung-untungan untuk menggunakan dengan benar.

Item 67. Avoid excessive synchronization (Hindari Terlalu Banyak Menggunakan Sinkronisasi)

Item sebelumnya membahas mengenai kurangnya sinkronisasi. Item ini fokus pada masalah yang berlawanan dengan itu. Tergantung situasi yang dihadapi, sinkronisasi yang terlalu sering bisa menurunkan performance, deadlock atau bahkan perilaku yang aneh.

Untuk menghindari liveness dan safety failures, jangan pernah menyerahkan ke client proses sinkronisasi method atau blok kode tertentu. Dengan kata

lain, dalam wilayah sinkronisasi, jangan menggunakan method yang dirancang untuk di-override, atau yang disediakan client dalam bentuk fungsi yang me-return value. Dari perspective kelas yang memiliki bagian sinkronisasi, method-method itu adalah alien. Kelas tidak memiliki pengetahuan tentang apa yang method itu lakukan dan tidak memiliki control atas itu. Bergantung pada apa yang sebuah method alien lakukan, memanggil itu dari bagian sinkronisasi bisa menyebabkan exception, deadlocks, atau data corruption.

```
public class ObservableSet<E> extends ForwardingSet<E> {

    public ObservableSet(Set<E> set) { super(set); }

    private final List<SetObserver<E>> observers = new
ArrayList<SetObserver<E>>();

    public void addObserver(SetObserver<E> observer) {

        synchronized(observers) {

            observers.add(observer);

        }

    }

    public boolean removeObserver(SetObserver<E> observer) {

        synchronized(observers) {

            return observers.remove(observer);

        }

    }

    private void notifyElementAdded(E element) {

        synchronized(observers) {

            for (SetObserver<E> observer : observers)

                observer.added(this, element);
```

```
            }
            // Alien method moved outside of synchronized block - open calls
            // private void notifyElementAdded(E element) {
            //       List<SetObserver<E>> snapshot = null;
            //       synchronized(observers) {
            //             snapshot = new
ArrayList<SetObserver<E>>(observers);
            //       }
            //       for (SetObserver<E> observer : snapshot)
            //             observer.added(this, element);
            //}
      }
      // cara yang lebih baik untuk konkuren koleksi.
      //private final List<SetObserver<E>> observers = new
CopyOnWriteArrayList<SetObserver<E>>();

      //public void addObserver(SetObserver<E> observer) {
      //       observers.add(observer);
      //}
      //public boolean removeObserver(SetObserver<E> observer) {
      //       return observers.remove(observer);
      //}
      //private void notifyElementAdded(E element) {
      //       for (SetObserver<E> observer : observers)
      //             observer.added(this, element);
```

```
        //}
        @Override
        public boolean add(E element) {
                boolean added = super.add(element);
                if (added)
                        notifyElementAdded(element);
                return added;
        }
        @Override
        public boolean addAll(Collection<? extends E> c) {
                boolean result = false;
                for (E element : c)
                        result |= add(element); // calls notifyElementAdded
                return result;
        }
}
```

Untuk lebih detail, please lihat kode program dan penjelasannya di Github.

Over-sinkronisasi memberikan banyak kelemahan. Membuat memori menjadi tidak konsisten, hal berikutnya adalah over-sinkronisasi membatasi kemampuan VM untuk optimalisasi code execution.

Kita seharusnya membuat kelas yang mutable bersifat thread safe jika itu dimaksudkan untuk penggunaan yang konkuren dan kita bisa meningkatkan konkurensi secara signifikan lebih baik dengan sinkronisasi internal daripada kita mengunci object dari luar. Sebaliknya, jangan lakukan sinkronisasi apapun. Biarkan client melakukan sinkronisasi eksternal yang dianggap

tepat. Pada awal java platform, banyak kelas melanggar aturan ini. Contohnya, StringBuffer hampir selalu digunakan oleh thread tunggal, namun mereka melakukan sinkronisasi internal. Untuk alasan ini bahwa StringBuffer digantikan oleh StringBuilder yang tidak melakukan sinkronisasi internal. Kalau ragu-ragu, jangan sinkronisasi kelas tertentu, tapi buat dokumentasi bahwa kelas itu tidak thread-safe.

Jika kita melakukan internal sinkronisasi, kita bisa melakukan sejumlah tehnik untuk meningkatkan konkurensi, seperti lock splitting(kunci saat memotong string), lock stripping (kunci saat stripping), dan tanpa blocking untuk control konkurensi.

Jika sebuah method memodifikasi sebuah static field, kita harus meng-snkronisasi akses ke field itu, bahkan jika method hanya digunakan oleh sebuah thread saja. Tidak mungkin bagi client untuk melakukan eksternal sinkronisasi pada method tertentu karena tidak ada kepastian bahwa client yang tidak ada hubungannya dengan itu akan melakukan-nya juga.

Simpulannya, untuk menghindari deadlock atau data corruption, jangan pernah panggil sebuah method alien di dalam wilayah sinkronisasi. Secara umum, coba untuk membatasi jumlah aktivitas yang dilakukan dari wilayah sinkronisasi. Ketika kita sedang mendesign sebuah kelas yang mutable, pikirkan tentang apakah itu seharusnya melakukan sinkronisasi sendiri. Di jaman multicore, itu lebih penting daripada sebelumnya untuk tidak menyinkronkan berlebihan. Sinkronkan kelas kita secara internal hanya jika ada alasan yang tepat untuk melakukan-nya, dan dokumentasikan pilihan dengan jelas.

Item 68. Prefer Executors and Tasks to Thread*.*

Pada release java 5, java.util.concurrent ditambahkan pada platform java. Package ini memiliki sebuah Executor Framework, yang merupakan interface fleksibel dalam fasilitas eksekusi kegiatan atau task. Membuat antrian kerjaan lebih baik dari pada pada versi java sebelumnya.

ExecutorService executor = Executors.newSingleThreadExecutor();

Here is how to submit a runnable for execution:

executor.execute(runnable);

And here is how to tell the executor to terminate gracefully (if you fail to do this, it is likely that your VM will not exit):

executor.shutdown();

Banyak hal yang bisa dilakukan dengan Executor Service.

1. kita bisa menunggu untuk menyelesaikan sebagian task tertentu.
2. kita juga bisa menunggu untuk beberapa atau semua task selesai (dengan *invokeAny atau invokeAll method*)
3. kita bisa menunggu sampai executor service menghentikan layanan ini dengan baik sampai selesai.
4. kita bisa membaca hasil dari tiap-tiap task setelah mereka selesai(dengan *ExecutorCompletionService*)

jika kita ingin lebih dari satu thread yang memproses request dari antrian, sederhananya panggil static factory yang berbeda untuk membuat executor service yang berbeda memanggil sebuah thread pool. Kita bisa membuat sebuah thread pool yang sesuai dengan sejumlah thread. Kelas java.util.concurrent.Executor memiliki sejumlah static factory yang menyediakan sejumlah executor yang pernah akan kita butuhkan. Jika, bagaimanapun, kita ingin beberapa hal yang keluar dari kebiasaan, kita bisa menggunakan ThreadPoolExecutor secara langsung. Kelas ini mengizinkan kita mengontrol hampir setiap aspek dati operasi yang dimiliki oleh thread pool.

Memilih menggunakan executor service untuk sebagian aplikasi sedikit tricky, Jika kita sedang menulis program yang berskala kecil, atau server dengan load yang ringan, menggunakan*executor.newCachedThreadPool*adalah secara umum pilihan yang baik, sebagaimana itu tidak membutuhkan konfigurasi dan umumnya telah melakukan semua dengan baik dan tepat.

Tapi, sebaliknya, cached thread pool adalah pilihan yang buruk untuk server dengan load data yang besar, dalam cached thread pool, meletakkan task yang tidak antri tapi segera dilepaskan ke thread untuk dieksekusi. Jika tidak ada satupun thread yang exist, maka sebuah thread baru akan diciptakan. Jika sebuah server terlalu berat yang mana CPU secara keseluruhan telah dipakai, dan lebih banyak task berdatangan, thread akan dibuat lebih banyak, yang akan membuat masalah semakin buruk. Oleh karena itu, untuk server dengan load data yang besar, kita lebih baik menggunakan*Executors.newFixedThreadPool*,yang akan memberikan kita sebuah pool dengan jumlah thread yang pasti, atau menggunakan kelas ThreadPoolExecutor secara langsung untuk control yang maximum.

Bukan hanya seharusnya kita menahan diri dari membuat antrian kerja sendiri, tapi kita harusnya menahan diri untuk bersentuhan secara langsung dengan thread. Kuncinya adalah bukan lagi Thread, yang menyajikan keduanya, unit kerja dan mekanisme untuk mengeksekusinya. Sekarang unit kerja dan mekanismenya adalah hal yang terpisah. Kuncinya adalah unit kerja, atau sering disebut task. Ada dua tipe task,*Runnable dan* sepupunya *Callable*, yang mirip seperti Runnable, tetapi berbentuk fungsi yang memngembalikan nilai. Mekanisme umum untuk eksekusi task adalah *Executor Service*jika kita berpikir tentang task dan mempersilahkan executor service mengeksekusi mereka untuk kita, kita diuntungkan dengan fleksibilitas yang luar biasa dalam menentukan kebijakan yang sesuai untuk eksekusi task yang kita punya. Esensinya, *Executor Framework* melakukan eksekusi apa yang *Collections Framework* lakukan untuk membuat koleksi atau proses mengumpulkan nilai dalam satu wadah.

Executor Framework juga memiliki sebuah pengganti untuk java.util.Timer, itu adalah*ScheduledThreadPoolExecutor*, untuk sementara lebih mudah menggunakan sebuah timer, sebuah scheduled thread pool executor jauh lebih fleksibel. Sebuah timer menggunakan hanya sebuah thread untuk eksekusi task, yang bisa melanggar akurasi waktu dalam task yang memiliki

waktu running panjang atau lama. Jika sebuah timer dengan satu thread mendapatkan exception yang fatal membuat crash, timer berhenti beroperasi. Sebuah scheduled thread pool executor mendukung multiple thread dan recover task dengan baik untuk task yang mendapatkan un-checked exception.

Treatment lengkap mengenai Executor Framework adalah melebihi cakupan dari buku ini, tapi jika pembaca tertarik, bisa langsung membaca buku *Java Concurency in Practice*.

Item 69. Prefer Concurrency Utilities to wait and notify.

Pada release java 5, java platform menyediakan tool konkurensi level tinggi yang melakukan sejumlah hal yang kita umumnya menggunakan wait dan notify. Daripada kesulitan dalam menggunakan wait dan notify dengan benar dan tepat, kita harusnya menggunakan tool konkurensi lebih tinggi. Tool konkurensi level tinggi dalam java.util.concurrent dibagi menjadi 3 kategori : Executor Framework yang telah dijelaskan pada chapter sebelumnya. Concurrent Collections dan Synchronizers. (keduanya akan dibahas pada chapter kali ini).

Koleksi yang bersifat konkuren memberikan kinerja yang sangat baik untuk beberapa koleksi struktur data yang standar, antara lain List, Queue, Map. Untuk memberikan tingkat konkurensi yang baik, mereka mengatur proses sinkronisasi secara internal. Oleh karena itu, sangat tidak mungkin untuk meniadakan aktifitas konkurensi dari sebuah koleksi yang bersifat konkuren, mengunci itu tidak akan memberikan efek yang baik, hanya membuat aplikasi berjalan lambat saja.

Ini berarti client tidak bisa secara details menyusun proses pemanggilan method untuk koleksi yang berjalan secara konkuren. Beberapa interface koleksi memiliki-nya karena itu diperluas dengan *modifikasi operasi state-dependent*, yang mana itu mengkombinasikan beberapa tipe data primitif ke dalam operasi tunggal yang atomic.

Contohnya,*ConcurrentMap*extends*Map*dan menambahkan beberapa method, termasuk*putIfAbsent(key, value)*, yang meng-insertkan value yang dipasangkan dengan key jika nilai absent terpenuhi atau true, dan mengembalikan nilai terakhir yang terkait dengan key itu, atau tidak mengembalikan nilai sama sekali atau null jika key-nya tidak ada. Ini membuat itu mudah untuk diimplementasikan secara thread-safe atau mengakomodir untuk multithread. Contohnya, method dibawah menggambarkan tentang kerja dari*method String intern() :*

```
// Concurrent canonicalizing map atop ConcurrentMap - not optimal

private static final ConcurrentMap<String, String> map = new ConcurrentHashMap<String, String>();

public static String intern(String s) {

        String previousValue = map.putIfAbsent(s, s);

        return previousValue == null ? s : previousValue;

}
```

Faktanya, kita bisa melakukan yang lebih baik lagi,*ConcurrentHashMap*adalah optimalisasi untuk method pengambilan nilai, misalnya*get*. Oleh karena itu, awalnya kita bisa memanggil menggunakan *method get*terlebih dahulu, selanjutnya ingin memanggil method*putIfAbsent*hanya jika itu memang diperlukan.

```
// Concurrent canonicalizing map atop ConcurrentMap - faster!

public static String intern(String s) {

        String result = map.get(s);

        if (result == null) {

                result = map.putIfAbsent(s, s);

                if (result == null)

                        result = s;
```

```
            }

        return result;

}
```

selain menawarkan konkurensi yang sangat baik, *ConcurrentHashMap* juga sangat cepat. Kecepatannya sekitar 6x lebih cepat setelah di optimasi ketimbang masih menggunakan String.intern(tapi perlu di-ingat, bahwa String.intern harus menggunakan semacam referensi lemah untuk menjaga memori tidak nge-leak). Kecuali kalau, kita memiliki alasan yang menarik untuk melakukan selain itu, menggunakan ConcurrentHashMap adalah pilihan untuk *Collections.synchronizedMap*or*Hashtable*. Sederhananya, mengganti cara lama dalam menggunakan sikronisasi map dengan cara konkurensi map bisa meningkatkan kinerja dari aplikasi yang berjalan konkuren secara dratis. Lebih umum lagi, menggunakan konkurensi collections adalah pilihan untuk eksternal sinkronisasi pada collections.

Beberapa jenis *collections interface* telah difasilitasi dengan *blocking operations,* yang menunggu atau mem-block sampai mereka dapat berhasil dilakukan. Sebagai contohnya, BlockingQueue extends Queue dan menambahkan beberapa method, termasuk take, yang me-remove dan mengembalikan nilai dari *head of queue,* sedang menunggu jika queue kosong. Ini mengizinkan *blocking queues* untuk digunakan sebagai *work queues* (juga dikenal sebagai *producer consumer queues),* yang mana untuk satu atau lebih *producer threads* melakukan *enqueue work items* and dari satu atau lebih *consumer threadsmelakukan dequeue* dan process items sebagaimana mereka jadi tersedia. Sebagaimana harapan kita, kebanyakan implementasi dari *Executor Service*, termasuk juga *ThreadPoolExecutor*, menggunakan sebuah *BlockingQueue.*

Synchromizers adalah objek yang mengaktifkan threads untuk menunggu thread yang lainnya, mengizinkan mereka meng-koordinasikan tugas-tugas mereka. *Synchronizers*yang paling umum digunakan adalah

CountDownLatch dan *Semaphore.*Yang kurang umum digunakan adalah *CyclicBarrier* dan *Exchanger.*

CountDownLatch adalah penyelaras tunggal yang mengizinkan satu atau lebih thread untuk menunggu satu atau lebih thread lainnya untuk melakukan task tertentu. Satu-satunya konstruktor bagi CountDownLatch adalah integer yang merupakan jumlah berapa kali countDown method harus di panggil di dalam slot-nya sebelum semua thread yang sedang menunggu diizinkan untuk di jalankan atau proceed.

Hal ini mengejutkan, bahwa mudah untuk membangun hal-hal yang berguna di atas dengan cara sederhana ini. Sebagai contohnya,

1. Misalkan kita ingin membangun sebuah framework sederhana untuk waktu pelaksanaan yang bersamaan dari suatu tindakan.
2. Framework terdiri dari sebuah method tunggal yang mengontrol sebuah executor untuk menjalankan sebuah tindakan tertentu, sebuah level konkurensi menggambarkan jumlah tindakan atau task yang harus dieksekusi secara bersamaan, dan sebuah runnable mengambarkan tindakan.
3. Semua thread sudah siap untuk melakukan task sebelum waktu dari thread dimulai (perlu untuk tepat waktu). Ketika thread yang terakhir siap untuk melakukan task-nya, waktu telah dimulai, membiarkan semua thread melakukan tasknya. Sesegera mungkin thread yang terakhir selesai untuk melakukan tugasnya atau tindakannya, waktu berhenti.

Langsung menerapkan logika ini dengan menggunaka wait() and notify() akan bisa sedikit dikatakan menyebabkan kekacauan, tapi mengejutkannya bisa dilakukan dengan CountDownLatch.

```
// Simple framework for timing concurrent execution

public static long time(Executor executor, int concurrency, final Runnable action)
throws InterruptedException {

    final CountDownLatch ready = new CountDownLatch(concurrency);
```

```
        final CountDownLatch start = new CountDownLatch(1);
        final CountDownLatch done = new CountDownLatch(concurrency);

        for (int i = 0; i < concurrency; i++) {
            executor.execute(new Runnable() {
                public void run() {
                    ready.countDown(); // Tell timer we're ready
                    try {
                        start.await(); // Wait till peers are ready
                        action.run();
                    } catch (InterruptedException e) {
                        Thread.currentThread().interrupt();
                    } finally {
                        done.countDown(); // Tell timer we're done
                    }
                }
            });
        }
        ready.await(); // Wait for all workers to be ready
        long startNanos = System.nanoTime();
        start.countDown(); // And they're off!
        done.await(); // Wait for all workers to finish
        return System.nanoTime() - startNanos;
    }
```

Catat bahwa method menggunakan 3 count down latches.

1. *ready*, digunakan untuk memberitahukan waktu dari thread kapan mereka siap.
2. Semua threads lalu menunggu latch kedua yang bernama start.
3. Ketika thread terakhir memanggil *ready.countDown*, timer thread merekam waktu mulai dan memanggil *start.countDown*, mengizinkan semua thread untuk di jalankan.
4. Kemudian timer thread menunggu latch ketiga bernama *done*, sampai thread terakhir selesai dengan tasknya, dan memanggil *done.countDown*. Segera setelah ini terjadi, thread timer aktif dan merekam waktu berakhirnya.

pembahasan yang sekarang kita lakukan adalah mengenai concurrency utilities, contohnya tiga countdown latches pada contoh sebelumnya bisa digantikan dengan cyclic barrier saja. Code yang dihasilkan bahkan lebih ringkas lagi, tapi itu lebih sulit untuk dijelaskan dan dimengerti. Untuk informasi lanjutan, silakan baca Java concurrency in Practice.

Ketika kita seharusnya selalu menggunakan concurrency utilities daripada wait and notify, kita harus me-maintain code yang menggunakan wait and notify. Method wait digunakan untuk mmebuat thread menunggu untuk beberapa kondisi tertentu. Itu mesti dipanggil di dalam block sinkronisasi yang mengunci objek dalam proses memanggil.

```
// The standard idiom for using the wait method
synchronized (obj) {
        while (<condition does not hold>)
                obj.wait(); // (Releases lock, and reacquires on wakeup)
        ... // Perform action appropriate to condition
}
```

selalu gunakan wait loop idiom untuk memanggil method wait(), jangan pernah memanggil-nya diluar loop. Process looping menyajikan proses testing untuk kondisi sebelum dan setelah menunggu.

Cek kondisi sebelum proses waiting dan skip process wait jika kondisi telah ditahan untuk memastikan kehidupan objek. Jika kondisi telah ditahan dan method notify telah dipanggil sebelum thread menunggu, tidak ada jaminan bahwa thread pernah lepas dari kondisi menunggu.

Cek kondisi setelah menunggu dan menuggu lagi jika kondisi perlu ditahan untk memastikan keamanan. Jika thread melakukan tindakan ketika kondisi tidak ditahan, itu bisa menghancurkan penjagaan yang dilakukan oleh kunci secara liar. Ada beberapa alasan sebuah thread harus aktif ketika kondisi tidak di tahan.

1. Thread yang lainnya bisa mengambil kunci dan menggantinya diantara waktu thread memanggil notify dan waktu menunggu thread aktif (not wait).
2. Thread yang lain bisa saja memanggil notify secara tidak sengaja ketika kondisi tidak di-hold.
3. Thread yang sedang menunggu bisa aktif padahal notify tidak pernah dipanggil, ini dikenal dengan *spurious wakeup.*

issue yang masih terkait adalah apakah kita akan menggunakan notify atau notifyAll untuk mengaktifkan thread yang sedang dalam kondisi wait. Notify hanya mangaktifkan single thread yang sedang menunggu, sedangkan notifyAll mengaktifkan semua thread yang sedang menunggu. Sering dikatakan bahwa kita seharusnya menggunakan notifyAll. Itu masuk akal, saran yang konservatif. Itu akan selalu memberikan hasil yang tepat, karena itu menjamin bahwa kita mengaktifkan semua thread yang perlu untuk diaktifkan. Kita mungkin akan mangaktifkan thread yang lainnya juga, tapi itu tidak akan mempengaruhi benarnya aplikasi kita berjalan. Tiap-tiap thread akan membaca kondisi untuk yang mana yang sedang menunggu dan menemukan itu salah dan akan menunggu lagi.

Sebagai sebuah optimasi, kita mungkin memilih untuk memanggil notify daripada notifyAll, jika semua thread bisa dikondisikan wait, buat mereka sedang waiting untuk kondisi yang sama dan hanya satu thread dalam satu waktu bisa bisa mendapatkan keuntungan dari kondisi menjadi true.

Simpulannya, menggunakan wait dan notify secara langsung seperti programming menggunakan bahasa mesin, sebagaimana kita bandingkan dengan bahasa level tinggi menyediakan library java.util.concurrent. Jaran kejadian, jika pernah alasan untuk menggunakan wait dan notify di kode program yang baru. Jika kita melakukan maintenance kode yang menggunakan wait dan notify, yakinkan bahwa itu selalu memanggil method wait dari dalam looping jenis while dengan menggunakan idiom yang standar. Method notifyAll seharusnya secara umum digunakan sebagai preferensi dari notify. Jika notify digunakan, kehati-hatian yang luar biasa diperlukan untuk memastikan aman dan objek tidak mati.

Item 70. Document thread safety.

Kelas yang menggunakan thread mesti didokumentasikan dalam artian orang yang menggunakan kelas itu di masa depan harus paham cara menggunakannya, karena kalau tidak dia akan membuat sejumlah asumsi yang sukur-sukur kalo benar, tapi kalau salah akan banyak masalah, sinkronisasi yang kacau, dan sinkronisasi yang berlebihan.

Kita mungkin sering mendengar untuk tahu bahwa kelas itu thread-safe atau tidak hanya dengan melihat modifier synchronized, tapi itu salah, untuk memastikan bahwa kelas itu bersifat konkuren dalam penggunaannya, sebuah kelas harus secara jelas mendokumentasikan level dari thread yang di-support olehnya.

Ada beberapa level of thread-safety. Dijelaskan secara lebih umum.

1. Immutable. Proses penciptaan dari kelas ini bersifat konstan. Tidak ada sinkronisasi eksternal yang diperlukan. Contohnya, String, Long, dan BigInteger.
2. Unconditionally thread safe - instance dari kelas ini bersifat mutable, tapi kelas memiliki sinkronisasi internal yang membuat instancenya bisa digunakan konkuren tanpa perlu sinkronisasi eksternal lagi. Contohnya, Random dan ConcurrentHashMap.
3. Conditionally thread safe - seperti point nomor 2, kecuali beberapa method memerlukan sinkronisasi eksternal untuk penggunaan konkurensi yang aman, Contohnya collections yang merupakan nilai balikan dari Collections.synchronized wrappers, yang iterasinya memerlukan eksternal sinkronisasi.
4. Tidak thread Safe - jenis instancenya adalah mutable. Untuk menggunakan secara konkuren sangat perlu eksternal sinkronisasi. Contohnya, ArrayList dan HashMap.
5. Thread hostile - kelas ini tidak aman untuk konkurensi, bahkan jika tiap proses dilakukan eksternal konkurensi, terjadi karena perubahan nilai pada static variable tanpa sinkronisasi, tapi untungnya sangat jarang penggunaannya, java hanya punya satu, itu juga sudah deprecated.(The System.runFinalizersOnExit)

   Semua kategori diatas, kecuali point nomor 5, dibahas pada*thread safety annotations in Java Concurrency in Pratice.* Yang mana mereka adalah Immutable, ThreadSafe dan NotThreadSafe, unconditionally dan conditionally thread safe dicover pada pembahasan ThreadSafe annotation.

   simpulannya, setiap kelas seharusnya didokumentasikan secara jelas apakah level supporting thread yang ada.

1. untuk conditionally thread safe, mesti jelas pemanggilan method mana yang memerlukan eksternal sinkronisasi.
2. untuk unconditionally thread safe, pertimbangkan untuk menggunakan private lock object di tempat sinkronisasi method. ini membantu melindungi kita dari gangguan sinkronisasi oleh client atau subclassnya dan memberikan

kita fleksibilitas untuk mengadopsi cara yang lebih tepat untuk control konkurensi pada release yang berikutnya.

```
// Private lock object idiom - thwarts denial-of-service attack

private final Object lock = new Object();

public void foo() {

        synchronized(lock) {

                …

        }

}
```

Item 71. Use Lazy Initialization Judiciously.

Maksud dari lazy initialization adalah menunda proses inisialisasi field sampe bener-benar dibutuhkan, kalo tidak pernah dibutuhkan, objek tidak akan di-inisialisasi. Walaupun ini merupakan proses optimasi, tetapi dapat membahayakan kelas dan instance initialization.

Nasehat yang paling bijak untuk lazy initialization adalah jangan lakukan itu kecuali benar-benar dibutuhkan. Dia adalah pedang bermata dua. Dia menurunkan cost untuk proses pembentukan instance tetapi juga meningkatkan cost untuk mengakses field yang bersifat seperti itu. Tergantung apa itu membutuhkan inisialisasi, berapa cost yang diperlukan untuk melakukan itu, seberapa sering lazy field diakses, lazy initialization bisa benar-benar membahayakan kinerja dari aplikasi.

Dalam sebagian besar keadaan, normal initialization lebih dianjurkan daripada lazy initialization.

```
// Normal initialization of an instance field

private final FieldType field = computeFieldValue();
```

If you use lazy initialization to break an initialization circularity, use a synchronized accessor, as it is the simplest, clearest alternative:

```
// Lazy initialization of instance field - synchronized accessor

private FieldType field;

synchronized FieldType getField() {

        if (field == null)

                field = computeFieldValue();

        return field;

}
```

lazy initialization bisa untuk static field dan juga instance field. Jika ingin menggunakan lazy initialization untuk kinerja dari static field, gunakan lazy initialization holder class idiom.

```
// Lazy initialization holder class idiom for static fields

private static class FieldHolder {

        static final FieldType field = computeFieldValue();

}

static FieldType getField() { return FieldHolder.field; }
```

Kalau ingin menggunakan lazy initialization untuk kinerja dari instance field, gunakan double-check idiom.

```
// Double-check idiom for lazy initialization of instance fields

private volatile FieldType field;

FieldType getField() {

        FieldType result = field;

        if (result == null) { // First check (no locking)

                synchronized(this) {
```

```
                result = field;
                if (result == null) // Second check (with locking)
                    field = result = computeFieldValue();
                }
            }
        return result;
}
```

idiom ini menghindari cost of locking ketika mengakses field setelah di-inisialisasi. Fokus di balik idiom ini adalah untuk mengecek nilai dari field sebanyak dua kali, pertama tanpa locking dan kemudian jika field muncul tanpa inisialisasi, kedua dengan locking. Hanya jika pengecekan kedua mengindikasikan bahwa field tidak di-inisialisasi, lalu kita inisialisasi field tersebut. Karena tidak ada locking jika field telah di-inisialisasi, kondisinya critical kenapa field dideklarasikan volatile.

Kode yang ada di atas mungkin terlihat sedikit sulit. Di beberapa bagian, kebutuhan terhadap local variabel result mungkin tidak terlihat dengan jelas. Apa yang variabel ini lakukan hanyalah memastikan bahwa field bersifat read only sekali di kondisi yang umum dimana dia telah di-inisialisasi. Meskipun tidak benar-benar diperlukan, ini mungkin meningkatkan performa dan lebih elegan berdasarkan standar yang diterapkan pada concurrent programming low-level. Pada komputer penulis, method di-atas lebih cepat 25% daripada version nyata yang tidak menggunakan local variabel.

Berdasarkan release sebelum java 5, idiom double-check tidak bekerja dengan handal karena*semantic of volatile modifier* tidak cukup handal untuk mendukung. Model dari memori yang diperkenalkan pada release 5 menyelesaikan problem ini. Hari ini, idiom double-check merupakan pilihan teknik untuk lazy initializing pada instance field. Ketika kita bisa menerapkan idiom double-check pada static method sama baiknya, tidak ada alasan

untuk melakukan juga, idiom kelas holder lazy initialization adalah pilihan yang lebih baik.

Dua jenis dari idiom double-check memberikan catatan. Kadang-kadang, kita mungkin membutuhkan lazy initialization terhadap instance field yang bisa mentolerir inisialisasi yang berulang. Jika kita bisa menemukan kondisi seperti itu, kita bisa menggunakan variant dari idiom double-check yang membuang second checking. Itu bukan kejutan, itu dikenal dengan *single check idiom.* Seperti terlihat di bawah ini.

```
private volatile FieldType field;

private FieldType getField() {

        FieldType result = field;

        if (result == null)

                field = result = computeFieldValue();

        return result;

}
```

Semua tehknik inisialisasi yang dijabarkan pada materi ini diterapkan untuk tipe data primitif dan sama baiknya untuk tipe data objek (object reference). Ketika double-check atau single-check idiom diterapkan untuk tipe data primitif numeric, nilai dari field di check ekivalen dengan 0 bukan null.

Jika kita tidak peduli apakah setiap thread melakukan hitung ulang nilai dari field, dan tipe data dari field adalah tipe data primitif selain, Long dan Double, kemudian kita mungkin memilih untuk membuang volatile modifier dari deklarasi field pada single check idiom. Variasi ini dikenal dengan nama*racy single-check idiom.* Itu mempercepat akses untuk beberapa arsitektur tertentu, dengan biaya tambahan untuk tiap tambahan inisialisasi, di atas satu thread yang mengakses-nya. Ini benar-benar teknik yang eksotik, bukan untuk penggunaan tiap kali. Ini, bagaimanapun digunakan oleh instance String untuk caching nilai dari hash code mereka.

Simpulannya, kita seharusnya meng-inisialisasi kebanyakan field secara normal, tidak lazy. Jika kita harus melakukan lazy inisialisasi dengan alasan untuk meningkatkan performa sebagai tujuan, atau untuk menghancurkan sebuah sirkulasi inisialisasi yang buruk, kemudian menggunakan teknik lazy initialization secara tepat. Untuk instance field, dia adalah*double-check idiom,* untuk static field, lazy initialization-nya adalah*idiom holder class.* Untuk instance field yang bisa mentolerir inisialiasi berulang, kita bisa mempertimbangkan penggunaan*single-check idiom.*

Item 72. Do not depend on thread scheduler

Ketika banyak thread yang mampu run, thread scheduler menentukan yang mana yang akan jalan dan untuk berapa lama. Beberapa sistem operasi yang normal akan mencoba untuk membuat penentuan itu secara fair/adil, tapi kebijakan bisa berbeda. Oleh karena itu, program yang ditulis dengan baik tidak seharusnya bergantung pada kebijakan sistem operasi. Setiap program yang bergantung pada thread scheduler untuk kebenaran, ketepatan dan performa adalah seperti menjadi tidak portable.

Cara terbaik untuk menulis program yang kuat, responsif dan protable adalah dengan memastikan bahwa jumlah rata-rata thread yang jalan tidak lebih besar secara significant dari jumlah processor. Ini meninggalkan thread scheduler dengan pilihan kecil, ini hanya menjalankan threads yang bisa jalan sampai mereka tidak bisa lagi jalan. Kebiasaan program tidak banyak berbeda, bahkan berbeda secara radikal dibawah kebijakan thread-scheduling. Catat bahwa jumlah dari thread yang bisa jalan tidak sama dengan jumlah totdal dari threads, yang bisa saja lebih besar. Thread yang sedang waiting bukan runnable.

Cara terbaik untuk menjaga jumlah dari thread yang siap jalan lebih rendah adalah dengan mengharuskan tiap-tiap thread melakukan beberapa task yang berguna dan kemudian menunggu arahan lebih jauh. Thread seharusnya tidak berjalan jika mereka tidak melakukan tindakan atau task

yang penting dan berguna. Dalam aturan dari Executor Framework, ini berarti menyediakan thread pool kita dengan tepat, dan menjaga task yang lumayan kecil dan layak dan independen antara satu dan yang lainnya. Task tidak seharusnya menjadi terlalu kecil atau dikirim secara berlebihan akan membahayakan performa.

Thread tidak seharusnya busy-wait, berulang melakukan pengecekan sebuah objek yang di-share sedang menunggu sesuatu terjadi. Disamping membuat program rentan terhadap liku0liku scheduler, busy-waiting meningkatkan secara significant kebutuhan untuk loading ke processor, mengurangi jumlah dari task yang penting dan berguna, yang thread lain bisa selesaikan. Contoh yang ekstrim tentang apa yang tidak boleh dilakukan perhatikan contoh kode program di bawah ini.

```
// Awful CountDownLatch implementation - busy-waits incessantly!
public class SlowCountDownLatch {
        private int count;
        public SlowCountDownLatch(int count) {
                if (count < 0)
                        throw new IllegalArgumentException(count + " < 0");
                this.count = count;
        }
        public void await() {
                while (true) {
                        synchronized(this) {
                                if (count == 0) return;
                        }
                }
```

```
    }
    public synchronized void countDown() {
        if (count != 0)
            count--;
        }
    }
```

di mesin penulis, SlowCountDownLatch sekitar 2000 kali lebih lambat daripada CountDownLatch ketika 1000 threads menunggu di latch. Ketika contoh ini mungkin tampak sedikit terlalu mengada-ada, ini tidak umum untuk melihat sistem dengan satu atau lebih threads yang perlu berjalan. Hasilnya mungkin tidak sedramatis SlowCountDownLatch, tapi performa dan portability seperti menderita.

Ketika berhadapan dengan program yang baru saja berjalan karena beberapa thread tidak mendapatkan waktu dari CPU untuk berjalan sebagaimana yang lainnya, hindari untuk tergoda yang menggunakan Thread.yield. Kita mungkin berhasil membuat program berjalan setelah terbentuk, tapi itu tidak akan bersifat portable. Pemanggilan method yield yang sama mungkin akan berhasil saat pertama kali, memburuk untuk kedua kali dan tidak memberikan dampak apapun untuk ketiga kali. Thread.yield belum teruji. Cara yang lebih baik adalah merestruktur aplikasi untuk mengurangi jumlah thread yang berjalan secara konkuren/bersama-sama.

Teknik yang berkaitan lainnya, adalah dengan menggunakan thread priorities. Thread priorities adalah yang paling tidak portable di antara banyak fitur pada platform java. Sangat tidak masuk akal untuk meningkatkan responsif sebuah aplikasi menggunakan sedikit thread priorities yang buruk, tapi itu jarang tepat dan tidak portable. Itu adalah hal yang tidak masuk akal untuk menyelesaikan masalah yang serius berkaitan dengan siklus hidup thread dengan menggunakan thread priorities sebagai

alat penyesuaian. Masalah akan muncul kembali sampai saat kita menemukan dan menyelesaikan penyebab sebenarnya.

Pada edisi pertama dari buku ini, telah dijelaskan bahwa penggunaan dari Thread.yield digunakan sebagian besar programmer hanya untuk testing. Ide itu muncul adalah untuk mencari bug yang muncul pada bagian-bagian kecil dari aplikasi. Teknik ini terbilang cukup efektif, tapi cara ini tidak dijamin bisa berjalan dengan baik. Oleh karena itu, kita seharusnya menggunakan Thread.sleep(1) daripada Thread.yield() untuk concurrent testing. Jangan menggunakan Thread.sleep(0), yang bisa selesai sangat cepat atau sesegera mungkin.

Simpulannya, jangan bergantung pada thread scheduler untuk kebenaran dari aplikasi kita. Hasil yang diberikan oleh aplikasi tidak robust/kuat dan tidak juga portable. Sebagaimana kewajaran, jangan menentukan Thread.yield atau prioritas thread. Fasilitas itu hanyalah merujuk pada scheduler. Thread priorities mungking digunakan sebagai pembanding untuk meningkatkan layanan pada program yang sudah berjalan, tapi kita seharusnya tidak pernah menggunakannya untuk memperbaiki program yang baru saja berjalan.

Item 73. Avoid thread groups

Sepanjang menggunakan thread, locks, dan monitor, abstraksi dasar yang ditawarkan oleh sistem threading adalah thread groups. Thread group adalah mekanisme yang berguna untuk mengisolasi applets untuk tujuan keamanan. Mereka tidak pernah benar-benar memenuhi janji ini, dan keamanan yang mereka katakan bahkan tidak pernah disebutkan di dalam modul java security sebagai standar keamanan.

Diberikan thread groups tidak menyediakan fungsionalitas dari keamanan untuk dibicarakan, apa fungsi keamanan yang mereka sediakan ? tidak banyak. Mereka mengizinkan kita untuk menerapkan Thread primitif untuk

semua thread sekaligus. Beberapa dari thread primitif itu telah deprecated dan reminder-nya tidak digunakan secara teratur.

Ironisnya, ThreadGroup API adalah kelemahan dalam standar keamanan thread. Untuk mendapatkan list thread yang eksis di thread group, kita harus memanggil method enumerate, yang akan meminta sebagai sebuah parameter panjang dari array yang cukup untuk menampung semua threads yang aktif. Method activeCount mengembalikan jumlah threads yang aktif dalam sebuah thread group, tapi tidak ada jaminan bahwa jumlah yang diberikan akan selalu akurat. Jika jumlah thread meningkat dan array masih terlalu kecil, method enumerate secara diam-diam tidak memperdulikan atau men-skip thread lainnya yang tidak memiliki ruang di objek array tersebut.

Thread group itu telah usang.

Simpulannya, thread group tidak menyediakan banyak cara untuk menggunakan fungsionalitasnya, dan banyak fungsionalitas yang disediakan adalah cacat. Thread groups adalah bukti nyata terbaik tentang exksperimen yang gagal, dan kita seharusnya dengan mudah mengabaikan kehadirannya. Jika kita merancang sebuah kelas yang berurusan dengan thread group yang logic, kita seharusnya menggunakan thread pool executor.

CHAPTER 11.

Serialization.

Chapter ini berfokus pada object serialization API, yang menyediakan sebuah frame work untuk meng-encoding object sebagai stream byte dan menyusun ulang objek itu dari byte stream. Encoding object ke byte stream dikenal sebagai serializing object, proses kebalikannya dikenal dengan deserializing. Sekali sebuah object telah di-serialize, encodingnya bisa di transfer dari satu VM to VM lainnya atau disimpan dalam disk untuk proses deserialization berikutnya. Serialization menyediakan standar reprensentasi untuk komunikasi remote, dan format standar persistent data untuk arsitektur

komponen java beans. Fitur yang layak untuk dicata di dalam chapter ini adalah serialization proxy pattern, yang bisa membantu kita untuk menghindari banyak perangkap dari serialisasi objek.

Item 74. Implement serializable judiciously

Membuat instance kelas menjadi serializable adalah semudah menambahkan kata "*implement serializable*" pada deklarasi kelasnya. Karena ini terlalu mudah dilakukan, ada sebuah persepsi yang kurang tepat bahwa serialization membutuhkan hanya sedikit usaha pada programmer. Kebenarannya jauh lebih kompleks. Ketika cost yang nyata atau segera dirasakan ketika membuat kelas menjadi serializable bisa dianggap sepele, cost yang jauh ke depan sering banyak, penting dan besar.

Cost utama dalam peng-implementasian Serializable adalah bahwa itu mengurangi fleksibilitas untuk mengganti implementasi dari kelas jika itu telah di-release. Ketika sebuah kelas meng-implement serializable, byte stream encodingnya menjadi bagian dari API yang di-eksport keluar. Sekali kita mendistribusikan sebuah kelas besar-besaran, kita harus mendukung bentuk serializable selamanya, seperti kita diminta untuk mendukung semua bagian lain dari API yang diekspor. Jika kita tidak mengusahakan untuk merancang serialized form secara custom, tapi menerima bentuk defaultnya, bentuk serializable akan menjadi bagian dari internal represntation yang asli selamanya. Dengan kata lain, jika kita menerima bentuk/rancangan default berupa serialized form, private kelas dan instance private package field menjadi bagian dari API yang di-eksport, dan dan praktek meminimalkan akses ke fields menghilangkan efektifitasnya sebagai alat untuk mengaburkan informasi tentang kelas yang isinya.

Jika kita menerima bentuk default dari serializable dan kemudian mengubah internal representation dari kelas itu, maka sebuah ketidaksesuaian perubahan pada serialized form akan muncul. Client akan melakukan serialisasi menggunakan format yang lama, dan melakukan deserialisasi

menggunakan format baru, maka akan menyebabkan kegagalan. Sebenarnya bisa mengubah format internal dengan *ObjectOutputStream.putFields*and *ObjectInputStream.readFields,* tapi itu akan sulit dan meninggalkan jejak pada source code. Oleh karena itu, kita harus merancang kode program untuk jangka panjang.

Ketika kita meng-implement serializable, maka secara otomatis akan diciptakan sebuah static final variabel bernama serialVersionUID, yang diciptakan sesuai dengan kondisi kelas kita, terkait pula dengan method dan field yang eksis. Jika kita coba mengganti tapi tidak sesuai dengan kondisi kelas kita, maka kita akan mendapatkan exception, namanya*InvalidClassException*saat runtime.

Kedua, cost yang harus dibayar ketika meng-implement serializable adalah mereka meningkatkan kemungkinan munculnya bug dan munculnya lubang keamanan. Objek lain umumnya memilikin konstruktor sendiri untuk menggunakannya, tetapi jika objek kita bersifat serializable, maka dia memiliki sebuah konstruktor lainnya yang tersembunyi dan tidak terlihat, yang tidak bisa dikontrol, inilah lubang yang bisa menjadi masalah.

Ketiga, cost yang harus dibayar setelah meng-implement serializable adalah meningkatkan beban pengujian terkait dengan merilis versi baru dari kelas. Ketika versi baru keluar, kita harus memastikan versi terbaru bisa di serialize dan versi lama bisa di deserialize, dan sebaliknya juga. Sehingga jumlah scenario testing yang diperlukan jadi makin banyak.

Mengimplementasikan interface Serializable bukan keputusan yang akan dilakukan dengan ringan. Kita harus menimbang antar baik dan buruk serta kegunaan dari kelas yang akan meng-implement seriailizable, jika data akan ditransmisikan, maka adalah tepat menggunakan serializable, dengan menimbang misalnya Date dan BigInteger seharusnya meng-implement serializable, karena mungkin akan di-serialize untuk disimpan atau dikirim

via jaringan, begitu juga dengan Objek yang akan digunakan untuk tipe collection.

Kelas yang dirancang untuk pewarisan seharunsya jarang meng-implement Serializable, dan interface seharusnya jarang meng-extend itu juga. Tapi ada kondisi yang membuat kita bisa melanggar anjuran ini, misalnya jika sebuah kelas ada di dalam framework yang mengharuskan semua anggotanya meng-implement atau meng-extends Serializable, maka masuk akal untuk melanggar anjuran diatas.

Kelas yang dirancang untuk pewarisan yang melakukan implement Serializable termasuk Throwable, Component,dan HttpServlet. Throwable meng-implement Serializable sehingga exception dari RMI bisa dilempar dari server ke client. Component meng-implement Serializable sehingga GUI bisa dikirim, disimpan, dan dipulihkan. HttpServlet meng-implement Serializable sehingga session bisa di-cache.

Jika kita meng-implement sebuah kelas dengan instance field yang adalah serializable dan extendable, ada sebuah peringatan yang mesti kita perhatikan. Jika sebuah kelas memiliki kemungkinan versi nilai yang akan dilanggar jika instancce field di-inisialisasi ke nilai default-nya(0 untuk tipe integer, false untuk tipe boolean dan null untuk tipe object reference), kita harus menambahkan method *readObjectNoData* ke dalam kelas itu.

```
// readObjectNoData for stateful extendable serializable classes

private void readObjectNoData() throws InvalidObjectException {

        throw new InvalidObjectException("Stream data required");

}
```

Kalo kita ingin tahu, method *readObjectNoData* telah ditambahkan pada release java 4 untuk membantu sebuah kasus yang melibatkan penambahan superclass dari serializable ke sebuah kelas serializable yang telah eksis. Detailsnya, dapat ditemukan pada Serializable Spesification.

Ada sebuah keberatan mengenai pilihan untuk tidak meng-implement serializable. Jika sebuah kelas dirancang untuk pewarisan yang tidak serializable, itu tidak mungkin untuk membuat subclassnya menjadi serializable. Khususnya, itu akan tidak mungkin jika superclass tidak menyediakan sebuah parameter konstruktor yang tidak bisa diakses. Oleh karena itu, kita seharusnya mempertimbangkan menyediakan parameter pada konstruktor untuk kelas yang tidak serializable yang dirancang untuk pewarisan. Biasanya, kita tidak perlu melakukan tindakan ini karena banyak kelas yang dirancang untuk pewarisan tidak memiliki state, tapi ini tidak selalu menjadi kasusnya.

```
// Nonserializable stateful class allowing serializable subclass

public abstract class AbstractFoo {

        private int x, y; // Our state

        // This enum and field are used to track initialization

        private enum State { NEW, INITIALIZING, INITIALIZED };

        private final AtomicReference<State> init = new
AtomicReference<State>(State.NEW);

        public AbstractFoo(int x, int y) { initialize(x, y); }

        // This constructor and the following method allow

        // subclass's readObject method to initialize our state.

        protected AbstractFoo() { }

        protected final void initialize(int x, int y) {

                if (!init.compareAndSet(State.NEW, State.INITIALIZING)) throw new
        IllegalStateException(

                        "Already initialized");

                        this.x = x;

                        this.y = y;
```

```
                ... // Do anything else the original constructor did
                init.set(State.INITIALIZED);
        }
        // These methods provide access to internal state so it can
        // be manually serialized by subclass's writeObject method.
        protected final int getX() { checkInit(); return x; }
        protected final int getY() { checkInit(); return y; }
        // Must call from all public and protected instance methods
        private void checkInit() {
            if (init.get() != State.INITIALIZED)
                throw new IllegalStateException("Uninitialized");
            }
        ... // Remainder omitted
}
```

Semua method yang ada di dalam kelas *AbstractFoo* yang memiliki identifier public dan protected harus memanggil method checkInit sebelum melakukan hal lain. Ini untuk memastikan bahwa proses pemanggilan method yang gagal dapat berakhir dengan cepat dan aman jika subclass yang buruk gagal untuk melakukan inisialisasi terhadap instance itu. Catat bahwa initialized field adalah sebuah AtomicReference (*java.util.concurrent.atomic.AtomicReference*). Ini diperlukan untuk memastikan integrity dari objek. Jika tidak ada peringatan dini tersebut, jika sebuah thread memanggil initialize pada saat thread kedua berusaha untuk menggunakan-nya, maka thread kedua akan mendapati instance dalam kondisi status yang inconsistent nilainya.

```
// Serializable subclass of nonserializable stateful class
```

```
public class Foo extends AbstractFoo implements Serializable {

        private void readObject(ObjectInputStream s) throws IOException,
ClassNotFoundException {

                s.defaultReadObject();

                // Manually deserialize and initialize superclass state

                int x = s.readInt();

                int y = s.readInt();

                initialize(x, y);

        }

        private void writeObject(ObjectOutputStream s) throws IOException {

                s.defaultWriteObject();

                // Manually serialize superclass state

                s.writeInt(getX());

                s.writeInt(getY());

        }

        // Constructor does not use the fancy mechanism

        public Foo(int x, int y) { super(x, y); }

        private static final long serialVersionUID = 1856835860954L;

}
```

Inner kelas tidak seharusnya meng-implement serializable. Mereka menggunakan pen-generate dari compiler, synthetic fields, untuk menyimpan reference untuk melampirkan instance dan untuk menyimpan nilai dari local variabel dari bagian lampiran. Bagaimana cara field-field ini dihubungkan dengan kelasnya tidak dijelaskan dengan rinci, sebagaimana namanya yang berarti anonim dan kelas local. Oleh karena itu, bentuk

default dari inner kelas yang serializable tidakk didefinisikan. Sebuah anggota kelas yang static bisa, bagaimana pun meng-implement serializable.

Simpulannya, pendapat mengenai mudah untuk meng-implement serializable terdengar munafik, Kecuali kalau sebuah kelas telah ditempatkan pada penggunaan jangka pendek saja, meng-implement serializable adalah sebuah keseriusan dan seharusnya di-ikuti dengan kepedulian. Peringatan lanjutan adalah jaminan jika sebuah kelas dirancang untuk pewarisan. Untuk beberapa kelas, sebuah design menengah diantara meng-implement serializable dan melarang-nya di dalam sub-kelas untuk menyediakan construktor yang memiliki parameter yang bisa di-akses. Cara ini di-ijinkan, tapi tidak diharuskan, sub-kelas untuk meng-implement serializable.

Item 75. Consider using a custom serialized form

Seperti telah dijelaskan di chapter sebelumnya, serializable tidak sesederhana kedengerannya, pastikan bahwa kita telah menggunakannya dengan tepat, karena kita tidak akan lolos dari masalah yang ditimbulkannya seandainya kita akan memodifikasi kelas tersebut di masa depan.

Jangan pernah menerima begitu saja, bentuk default dari serializable tanpa memikirkan apakah itu tepat atau tidak untuk kelas tersebut, bukan hanya untuk hari ini atau kali ini, tetapi jauh ke depan, ketika kita akan melakukan maintenance dan modifikasi beberapa bagian yang terkait dengan kelas itu. Pertimbangkan beberapa hal, fleksibilitas, performance atau kinerja, dan kebenaran dan ketepatan. Penggunaan bentuk default dari object serializable mungkin akan tepat jika representasi fisik obyek identik dengan logical content. Sebagai contoh, bentuk defauilt serializable akan masuk akal digunakan untuk kelas di bawah ini.

```
// Good candidate for default serialized form

public class Name implements Serializable {

        /**
```

```
     * Last name. Must be non-null.
     * @serial
     */
     private final String lastName;
     /**
     * First name. Must be non-null.
     * @serial
     */
     private final String firstName;
     /**
     * Middle name, or null if there is none.
     * @serial
     */
     private final String middleName;
     ... // Remainder omitted
}
```

Logic berbicara bahwa nama terdiri dari tiga bagian, first nama, middle name, dan last name. Kenyataan yang terlihat di kelas Nama persis sama dengan logic-nya. Wujud fisik kelas persis sama dengan common logical ketika kita berbicara tentang nama.

Bahkan jika kita yakin dan memutuskan bahwa bentuk default serializable adalah pilihan yang tepat, kita harus memastikan bahwa kita menyediakan method readObject untuk meyakinkan menerima berbagai tipe dan bentuk nilai juga tentang keamanan. Dalam kasus nama, method readObject memastikan bahwa nilai dari first name dan last name bukan null.

Contoh berikutnya adalah contoh yang salah.

```
// Awful candidate for default serialized form

public final class StringList implements Serializable {

        private int size = 0;

        private Entry head = null;

        private static class Entry implements Serializable {

                String data;

                Entry next;

                Entry previous;

        }

        ... // Remainder omitted

}
```

Lupakan sejenak tentang cara implementasi List String yang sudah baik, ini hanya contoh untuk menjabarkan masalah dan cara penyelesaiannya.

Logic yang kita peroleh adalah, kelas merepresentasikan tentang urutan string. Secara fisik merepresentasikan urutan sebagai double linked list. Jika kita menerima bentuk default dari serializable, bentuk default serializable akan dengan susah payah memapping setiap nilai entry dalam Linked List dengan semua link diantara nilai entri tersebut, secara bolak-balik atau dua arah.

Menggunakan bentuk default serializable ketika representasi fisik dan logicnya berbeda memiliki 4 kelemahan.

1. Secara permanen mengikat API yang dieksport keluar dengan representasi internal. Pada contoh diatas, inner kelas Entry menjadi bagian dari public API. Jika representasi berubah pada release berikutnya, kelas StringList masih harus menerima jenis linked list saat input dan men-generate output juga.

Kelas ini tidak akan pernah lepas dari semua kode terkair linked list entri, meskipun itu sudah tidak digunakan lagi.

2. Perlu space memori yang banyak dan cenderung berlebihan.
3. Perlu space waktu yang banyak dan cenderung berlebihan, pemprosesan membuthkan waktu lebih lama dan panjang.
4. Bisa menyebabkan stack overflows.

Bentuk kelas StringList dengan custom serializable.

```
// StringList with a reasonable custom serialized form

public final class StringList implements Serializable {

        private transient int size = 0;

        private transient Entry head = null;

        // No longer Serializable!

        private static class Entry {

                String data;

                Entry next;

                Entry previous;

        }

        // Appends the specified string to the list

        public final void add(String s) { ... }

        /**

        * Serialize this {@code StringList} instance.

        *

        * @serialData The size of the list (the number of strings

        * it contains) is emitted ({@code int}), followed by all of

        * its elements (each a {@code String}), in the proper

        * sequence.
```

```
     */
     private void writeObject(ObjectOutputStream s) throws IOException {
          s.defaultWriteObject();
          s.writeInt(size);
          // Write out all elements in the proper order.
          for (Entry e = head; e != null; e = e.next)
               s.writeObject(e.data);
     }
     private void readObject(ObjectInputStream s) throws IOException,
ClassNotFoundException {
          s.defaultReadObject();
          int numElements = s.readInt();
          // Read in all elements and insert them in list
          for (int i = 0; i < numElements; i++)
               add((String) s.readObject());
     }
     ... // Remainder omitted
}
```

Jika semua instance field bersifat transient, secara teknikal dibolehkan untuk membuang bagian yang memanggil *defaultWriteObject* dan *defaultReadObject*, tapi itu tidak direkomendasikan. Bahkan jika semua instance field adalah bentuk transient, memanggil defaultWriteObject mempengaruhi bentuk default dari serializable, sehingga sangat meningkatkan fleksibiltas. Bentuk serializable yang dihasilkan memungkinkan untuk menambahkan instance field non-transient pada release berikutnya sambil tetap mempertahankan tingkat compatible antara

versi lama dan versi barunya. Jika sebuah instance merupakan serializable pada versi kemudian dan bukan serializable pada versi sebelumnya, field tambahan akan diabaikan. Versi sebelumnya yang memiliki readObject akan gagal memanggil defaultReadObject, dan proses deserialization akan karena StreamCorruptedException.

Sebelum memutuskan untuk membuat field menjadi non-transient, pastikan bahwa nilai tersebut merupakan bagian dari logical state sebuah objek. Jika kita menggunakan bentuk default serializable, kebanyakan atau semua instance field seharusnya diberi label transient, sebagaimana contoh pada StringList di atas.

Jika kita menambahkan modifier transient pada bentuk default dari serializable. ingatlah bahwa instance field akan diberi nilai default saat inisialisasi ketika instance ada pada fase deserialisasi, null untuk object reference, 0 untuk tipe data primitif yang berbentuk numeric, dan false untuk tipe data boolean. Jika nilai-nilai ini tidak diterima pada beberapa field bersifat transient, kita mesti menyediakan sebuah readObject yang akan memanggil defaultReadObject method dan kemudian membuat ulang field transient untuk menerima nilai-nya. Alternatifnya, tiap-tiap field ini bisa menjadi lazy inisialisasi saat pertama kali digunakan.

Apakah menggunakan bentuk default dari serializable atau tidak, kita harus memaksakan beberapa sinkronisasi pada serialization objek bahwa kita akan memaksakan sinkronisasi untuk beberapa method yang membaca keseluruhan objek.

Kita harus membuatnya menjadi thread-safe.

```
// writeObject for synchronized class with default serialized form

private synchronized void writeObject(ObjectOutputStream s) throws IOException
{

        s.defaultWriteObject();
```

```
}
```

jika kita menambahkan sinkronisasi dalam writeObject method, kita harus memastikan bahwa dia mengikuti standar ordering kunci seperti yang lainnya, atau kita beresiko mengalami deadlock ketika meminta pergantian akses resource tertentu.

Tanpa memperhatikan jenis serializable apa yang kita pilih, deklarasikan sebuah serial UID secara explisit pada setiap kelas yang meng-implement serializable yang kita tulis. Jika kita tidak menambahkan itu, akan ada issue terkait performance, karena diperlukan proses komputasi yang lumayan lama saat runtime untuk men-generate UID.

1. Kita bisa men-generate UID dengan menggunakan *Serialver* utility.
2. Kita bisa juga menggunakan serial UID yang telah eksis untuk kelas yang baru, tinggal ambil saja, generate UID dengan *Serialver* untuk versi yang lama.
3. Kita bisa juga mengganti UID pada existing kelas jika ingin membuat kelas yang baru dibuat tidak kompatible dengan kelas yang lama. Kita akan mendapat InvalidClassException jika ingin melakukan deserialisasi pada instance kelas yang sudah berbentuk serializable.

Simpulan, ketika kita telah memutuskan bahwa kelas tertentu seharusnya berbentuk serializable, pikirkan dengan serius apa bentuk serializable yang tepat. Menggunakan bentuk default dari serializable hanya jika itu mendeskripsikan logical state dari objek yang bersangkutan, cara lain adalah dengan mendesign custom serializable format. Kita seharusnya menyediakan cukup banyak waktu untuk merancang bentuk dari serializable sebuah kelas, sebagaimana kita menyediakan waktu untuk merancang sebuah exported method. Sebagaimana kita tidak bisa membuang begitu saja exported method pada saat release berikutnya, kita tidak bisa remove field dari bentuk default serializable, mereka harus dijaga agar tetap kompatible untuk versi serializable selamanya. Memilih bentuk serializable

yang salah bisa menyebabkan masalah yang permanen, efek buruk pada kompleksitas dan kinerja dari kelas.

Item 76. Write readObject method defensively.

```
// Immutable class that uses defensive copying

public final class Period {

        private final Date start;

        private final Date end;

        /**

        * @param start the beginning of the period

        * @param end the end of the period; must not precede start

        * @throws IllegalArgumentException if start is after end

        * @throws NullPointerException if start or end is null

        */

        public Period(Date start, Date end) {

                this.start = new Date(start.getTime());

                this.end = new Date(end.getTime());

                if (this.start.compareTo(this.end) > 0) throw new
        IllegalArgumentException(

                        start + " after " + end);

        }

        public Date start () { return new Date(start.getTime()); }

        public Date end () { return new Date(end.getTime()); }

        public String toString() { return start + " - " + end; }

        ... // Remainder omitted
```

```
}
```

Dugaannya kita memutuskan bahwa kita ingin kelas ini menjadi serializable, karena ada kesesuaian antara logical content dan representasi fisiknya. Tapi tidak masuk akal untuk melakukan itu. Bagaimanapun tidak ada jaminan untuk meng-handle kondisi kritis saat field value tidak sesuai atau tidak compatible.

Masalahnya terletak pada kondisi method *readObject* yang sebenarnya sama dengan public konstruktor, dan itu menuntut treatment yang sama seperti konstruktor lain. Sama seperti konstruktor harus memeriksa argumen untuk kevalidan dan membuat copy objek yang defensif secara tepat.

Sederhananya, sebuah method readObject adalah konstruktor yang membaca kumpulan byte stream dari parameter. Pada kondisi normal, byte stream akan dibaca dan disusun ulang menjadi sebuah instance objek. Kondisi menjadi rawan seandainya byte stream yang digunakan untuk meng-create ulang instance objek tidak cocok terhadap kelas itu. Seandainya kita memaksakan untuk membuat kelas Period menjadi serializable, maka implementasi dari kelas itu akan terlihat buruk seperti contoh di bawah.

```
public class BogusPeriod {

// Byte stream could not have come from real Period instance!

private static final byte[] serializedForm = new byte[] {

        (byte)0xac, (byte)0xed, 0x00, 0x05, 0x73, 0x72, 0x00, 0x06,

        0x50, 0x65, 0x72, 0x69, 0x6f, 0x64, 0x40, 0x7e, (byte)0xf8,

        0x2b, 0x4f, 0x46, (byte)0xc0, (byte)0xf4, 0x02, 0x00, 0x02,

        0x4c, 0x00, 0x03, 0x65, 0x6e, 0x64, 0x74, 0x00, 0x10, 0x4c,

        0x6a, 0x61, 0x76, 0x61, 0x2f, 0x75, 0x74, 0x69, 0x6c, 0x2f,

        0x44, 0x61, 0x74, 0x65, 0x3b, 0x4c, 0x00, 0x05, 0x73, 0x74,

        0x61, 0x72, 0x74, 0x71, 0x00, 0x7e, 0x00, 0x01, 0x78, 0x70,
```

```
        0x73, 0x72, 0x00, 0x0e, 0x6a, 0x61, 0x76, 0x61, 0x2e, 0x75,
        0x74, 0x69, 0x6c, 0x2e, 0x44, 0x61, 0x74, 0x65, 0x68, 0x6a,
        (byte)0x81, 0x01, 0x4b, 0x59, 0x74, 0x19, 0x03, 0x00, 0x00,
        0x78, 0x70, 0x77, 0x08, 0x00, 0x00, 0x00, 0x66, (byte)0xdf,
        0x6e, 0x1e, 0x00, 0x78, 0x73, 0x71, 0x00, 0x7e, 0x00, 0x03,
        0x77, 0x08, 0x00, 0x00, 0x00, (byte)0xd5, 0x17, 0x69, 0x22,
        0x00, 0x78 };

    public static void main(String[] args) {
        Period p = (Period) deserialize(serializedForm);
        System.out.println(p);
    }
    // Returns the object with the specified serialized form
    private static Object deserialize(byte[] sf) {
        try {
            InputStream is = new ByteArrayInputStream(sf);
            ObjectInputStream ois = new ObjectInputStream(is);
            return ois.readObject();
        } catch (Exception e) {
            throw new IllegalArgumentException(e);
        }
    }
}
```

jika program di-atas jalan dan running, maka kita akan memperoleh masalah, karena format byte stream yang disajikan tidaklah benar.

Untuk menyelesaikan masalah ini, kita mesti membuat method readObject yang memiliki defaultReadObject method di dalamnya dan kemudian mengecek kebenaran objek yang akan di-deserialisasi. Jika checking gagal, maka kita akan men-throws InvalidObjectException, untuk mencegah proses deserialisasi sampai complete.

```
// readObject method with validity checking

private void readObject(ObjectInputStream s) throws IOException,
ClassNotFoundException {

        s.defaultReadObject();

        // Check that our invariants are satisfied

        if (start.compareTo(end) > 0)

                throw new InvalidObjectException(start +" after "+ end);

}
```

Ada masalah yang lebih halus masih mengintai. Ada kemungkinan untuk membuat mutable instance terhadap Period dengan menggunakan byte stream yang di create pakai konstruktor Period yang valid, kemudian ditambahkan extra reference ke field Date yang bersifat private dalam instance Period.

Penyerang membaca instance Period dari ObjectInputStream dan kemudian membaca celah objek reference yang ditambahkan ke dalam stream. Reference ini memberikan penyerang access ke objek reference melalui private Date field dalam objek Period.

Dengan membuat instances Date mutable, penyerang bisa mengubah instance dari Period. Kelas di bawah ini menjabarkan tentang itu.

```
public class MutablePeriod {
```

```
// A period instance
public final Period period;
// period's start field, to which we shouldn't have access
public final Date start;
// period's end field, to which we shouldn't have access
public final Date end;
public MutablePeriod() {
        try {
                ByteArrayOutputStream bos = new ByteArrayOutputStream();
                ObjectOutputStream out = new ObjectOutputStream(bos);
                // Serialize a valid Period instance
                out.writeObject(new Period(new Date(), new Date()));
                /*
                * Append rogue "previous object refs" for internal
                * Date fields in Period. For details, see "Java
                * Object Serialization Specification," Section 6.4.
                */
                byte[] ref = { 0x71, 0, 0x7e, 0, 5 }; // Ref #5
                bos.write(ref); // The start field
                ref[4] = 4; // Ref # 4
                bos.write(ref); // The end field
                // Deserialize Period and "stolen" Date references
                ObjectInputStream in=new ObjectInputStream(new
                ByteArrayInputStream(bos.toByteArray()));
```

```
                period = (Period) in.readObject();
                start = (Date) in.readObject();
                end = (Date) in.readObject();
            } catch (Exception e) { throw new AssertionError(e);
            }
        }
}
```

Untuk melihat dimana celahnya, perhatikan kode program di bawah ini.

```
public static void main(String[] args) {
        MutablePeriod mp = new MutablePeriod();
        Period p = mp.period;
        Date pEnd = mp.end;
        // Let's turn back the clock
        pEnd.setYear(78);
        System.out.println(p);
        // Bring back the 60s!
        pEnd.setYear(69);
        System.out.println(p);
}
```

Kalau program itu dijalankan, akan menghasilkan output seperti di bawah ini.

```
Wed Apr 02 11:04:26 PDT 2008 - Sun Apr 02 11:04:26 PST 1978
Wed Apr 02 11:04:26 PDT 2008 - Wed Apr 02 11:04:26 PST 1969
```

Jika kita lihat, nilai dari tahun dapat berubah, itu dikarenakan kita membuat Period yang mutable, tingkat immutability sebuah kelas memberikan pengaruh pada keamanan terhadap penyerang.

Tapi, akar masalah yang sebenarnya adalah karena method readObject tidak melakukan sebuah defensif copy. Ketika sebuah objek di deserialisasi, sangat urgent untuk melakukan defensive copy terhadap semua field yang membawa reference objek tersebut yang client seharusnya tidak bisa mengakses reference itu. Oleh karena itu, setiap kelas yang meng-implement serializable yang bersifat immutable tetapi memiliki atribut kelas yang bersifat mutable harus melakukan defensif copy field itu di dalam method readObject.

```
// readObject method with defensive copying and validity checking

private void readObject(ObjectInputStream s) throws IOException,
ClassNotFoundException {

        s.defaultReadObject();

        // Defensively copy our mutable components

        start = new Date(start.getTime());

        end = new Date(end.getTime());

        // Check that our invariants are satisfied

                if (start.compareTo(end) > 0)

                        throw new InvalidObjectException(start +" after "+ end);

}
```

Catat, bahwa defensif copy yang dilakukan sebelumnya untuk mengecek validitas dan itu tidak menggunakan method clone milik objek Date untuk melakukan defensif copy. Keduanya cara itu dibutuhkan untuk melindungi objek Period dari serangan. Catat juga, bahwa defensif copy tidak mungkin dilkaukan untuk objek dengan final modifier. Untuk menggunakan method

readObject, kita harus membuat field start and end bersifta non-final. Ini disayangkan, tapi ini lebih rendah dari dua masalah itu. Dengan method readObject yang baru dan membuat final modifier dari field start dan end. Kelas MutablePeriod diterjemahkan menjadi tidak efektif. Serangan di atas membuat output seperti di bawah.

Wed Apr 02 11:05:47 PDT 2008 - Wed Apr 02 11:05:47 PDT 2008

Wed Apr 02 11:05:47 PDT 2008 - Wed Apr 02 11:05:47 PDT 2008

Pada release java 4, method writeUnshared dan readUnshared ditambahkan pada ObjectOutputStream dengan tujuan mencegah serangan menggunakan object reference tanpa perlu melakukan defensif copy. Sayangnya, method itu rentan terhadap serangan canggih yang sama secara natural. Jangan menggunakan method writeUnshared dan readUnshared. Mereka secara teknikal lebih cepat daripada defensif copy, tapi mereka tidak menyediakan jaminan kemanan yang diperlukan.

Ini adalah sebuah test yang simple untuk memutuskan apakah default method readObject bisa diterima sebuah kelas, akankah kita merasa nyaman menambahkan sebuah konstruktor bersifat public yang ditempatkan sebagai wadah nilai parameter untuk tiap-tiap field yang non-transient di dalam objek dan menyimpan nilai di dalam field tanpa validation ? jika tidak, kita harus menyediakan method readObject, dan itu mesti melakukan validasi terhadap parameter dan defensif copy yang diperlukan oleh konstruktor. Alternatifnya, kita bisa menggunakan *serialization proxy pattern.*

Ada satu kesamaan antara readObject method dengan konstruktor, fokuskan pada kelas non final yang meng-implement serializable. Sebuah method readObject tidak harus memanggil sebuah method yang bisa di-override, secara langsung atau tidak langsung. Jika aturan ini dilanggar dan method di-override. method yang di-override akan jalan sebelum subclass-nya mengalami deserialisasi. Sebuah kegagalan kemungkinan yang akan menjadi hasil akhirnya.

Simpulannya, setiap saati kita membuat readObject method, adopsi pikiran bahwa kita sedang menulis public kontruktor yang harus memberikan instance yang valid mengenai apa saja byte stream yang diberikan. Jangan pernah berasumsi bahwa byte stream merepresentasikan sebuah instance serialisasi yang sebenarnya. Sementara, contoh pada bahasan ini fokus pada kelas yang menggunakan bentuk default serializable, semua issue yang diangkat berlaku untuk kelas dengan bentuk custom serializable.

Pedoman untuk menulis method readObject.

1. Untuk kelas dengan tipe object sebagai fieldnya, haruslah private, lakukan defensif copy untuk masinng-masing object itu. Mutabel komponen atau immutable komponen termasuk kategori ini.
2. Lakukan pengecekan untuk semua invariant dan leparkan sebuah exception, InvalidObejctException jika pengecekan gagal, pengecekan seharusnya melakukan defensif copy.
3. Jika seluruh object harus divalidasi setelah deserialisasi, gunakan ObjectInputValidation interface.
4. Jangan panggil semua method hasil override di dalam kelas, langsung ataupun tidak langsung.

Item 77. For instance control, prefer enum types to readResolve.

```
public class Elvis {

        public static final Elvis INSTANCE = new Elvis();

        private Elvis() { ... }

        public void leaveTheBuilding() { ... }

}
```

Bentuk diatas adalah singleton, memastikan hanya satu jenis objek itu yang diciptakan.

Perlu di-ingat, jika objek ini di-implement serializable, object ini tidak lagi singleton, entah itu menggunakan default atau custom serializable. Method

readObject apa saja, apakah itu explicit/custom atau default readObject, mengembalikan sebuah instance yang baru, yang tidak akan sama dengan yang pernah diciptakan sebelumnya.

Method readResolve mengizinkan kita mengganti objek yang lama dengan yang diciptakan oleh readObject. Jika kelas dari sebuah objek yang telah di-deserialisasi mendefinisikan deklarasi readResolve dengan tepat, method ini dipanggil pada objek baru setelah proses deserialisasi terjadi. Reference dari objek ini adalah objek yang baru diciptakan itu. Pada umumnya penggunaan method ini, tidak ada referensi ke objek yang baru dibuat untuk dipertahankan. sehingga segera memenuhi syarat untuk masuk garbage collection.

Method di bawah ini contoh readResolve untuk kelas Elvis.

```
// readResolve for instance control - you can do better!
private Object readResolve() {
        // Return the one true Elvis and let the garbage collector
        // take care of the Elvis impersonator.
        return INSTANCE;
}
```

method ini mengabaikan objek deserialisasi, mengembalikan instance Elvis yang sama dengan yang ada ketika proses inisialisasi. Oleh karena itu, bentuk default dari instance Elvis tidak perlu mengandung data yang real, semua instance field seharusnya dideklarasikan secara transient. Faktanya, jika kita bergantung pada readResolve, sebagai pengontrol instance, semua instance dari field dengan tipe objek harus dideklarasikan sebagai transient. Sebaliknya, sangan mungkin untuk penyerang yang kuat, untuk melindungi sebuah reference ke object deserialisasi sebelum method readResolve jalan, menggunakan teknik yang mirip-mirip dengan penyerangan pada MutablePeriod.

Lihat gimana caraya bekerja.

1. Buat kelas Stealer yang memiliki sepasang method readResolve dan sebuah instance field yang mengarah ke objek singleton yang serializable di dalam kelas Stealer yang tersembunyi. Dalam aliran serialisasi, gantikan singleton yang tidak transient dengan instance dari Stealer. Kita sekarang memiliki siklusnya, singleton mengandung stealer dan stelaer merefer atau mengarah pada singleton
2. Kerena singleton memiliki stealer, method readResolve yang dimiliki stealer akan jalan pertama ketika singleton di-deserialisasi. Sebagai hasilnya, ketika method readResolve yang di kelas Stelaer jalan, instance field masih merefer atau mengarah pada bagian singleton yang di-deserialisasi.
3. Method readResolve yang ada pada stealer meng-copy reference dari instance field ke static field, sehingga reference itu bisa diakses setelah readResolve jalan. Lalu, method mengembalikan nilai yang benar untuk field yang sedang tersembunyi. Jika dia tidak melakukan ini, VM akan melempar ClassCastException ketika sistem serialisasi mencoba untuk menyimpan reference dari stealer ke field itu.

Agar terlihat nyata, lihatlan singleton yang gagal ini.

```
// Broken singleton - has nontransient object reference field!

public class Elvis implements Serializable {

        public static final Elvis INSTANCE = new Elvis();

        private Elvis() { }

        private String[] favoriteSongs = { "Hound Dog", "Heartbreak Hotel" };

        public void printFavorites() {

                System.out.println(Arrays.toString(favoriteSongs));

        }

        private Object readResolve() throws ObjectStreamException {

        return INSTANCE;
```

```
        }
}
public class ElvisStealer implements Serializable {
        static Elvis impersonator;
        private Elvis payload;
        private Object readResolve() {
                // Save a reference to the "unresolved" Elvis instance
                impersonator = payload;
                // Return an object of correct type for favorites field
                return new String[] { "A Fool Such as I" };
        }
        private static final long serialVersionUID = 0;
}
public class ElvisImpersonator {
        // Byte stream could not have come from real Elvis instance!
        private static final byte[] serializedForm = new byte[] {
                (byte)0xac, (byte)0xed, 0x00, 0x05, 0x73, 0x72, 0x00, 0x05,
                0x45, 0x6c, 0x76, 0x69, 0x73, (byte)0x84, (byte)0xe6,
                (byte)0x93, 0x33, (byte)0xc3, (byte)0xf4, (byte)0x8b,
                0x32, 0x02, 0x00, 0x01, 0x4c, 0x00, 0x0d, 0x66, 0x61, 0x76,
                0x6f, 0x72, 0x69, 0x74, 0x65, 0x53, 0x6f, 0x6e, 0x67, 0x73,
                0x74, 0x00, 0x12, 0x4c, 0x6a, 0x61, 0x76, 0x61, 0x2f, 0x6c,
                0x61, 0x6e, 0x67, 0x2f, 0x4f, 0x62, 0x6a, 0x65, 0x63, 0x74,
                0x3b, 0x78, 0x70, 0x73, 0x72, 0x00, 0x0c, 0x45, 0x6c, 0x76,
```

```
            0x69, 0x73, 0x53, 0x74, 0x65, 0x61, 0x6c, 0x65, 0x72, 0x00,
            0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x02, 0x00, 0x01,
            0x4c, 0x00, 0x07, 0x70, 0x61, 0x79, 0x6c, 0x6f, 0x61, 0x64,
            0x74, 0x00, 0x07, 0x4c, 0x45, 0x6c, 0x76, 0x69, 0x73, 0x3b,
            0x78, 0x70, 0x71, 0x00, 0x7e, 0x00, 0x02
        };
        public static void main(String[] args) {
            // Initializes ElvisStealer.impersonator and returns
            // the real Elvis (which is Elvis.INSTANCE)
            Elvis elvis = (Elvis) deserialize(serializedForm);
            Elvis impersonator = ElvisStealer.impersonator;
            elvis.printFavorites();
            impersonator.printFavorites();
        }
}
```

Hasilnya,

[Hound Dog, Heartbreak Hotel]

[A Fool Such as I]

Running program ini memberikan output di atas, membuktikan bahwa mungkin untuk menciptakan dua Instance elvis yang berlainan.

Kita bisa menyelesaikan permasalahan ini denga mendeklarasikan favorites sebagai transient, tapi lebih baik jika kita menjadikan Elvis sebagai element enum tunggal.

Sejarahnya, method readResolve digunakan untuk mengontrol semua instances yang serializable. Pada release java 5, ini tidak menjadi cara terbaik lagi dalam mengontrol instance kelas yang serializable. Sebagaimana telah didemokan oleh ElvisStealer, teknik ini rapuh dan memerlukan kepedulian yang besar.

Jika kita menjadikan Elvis sebuah enum, kita mendapat jaminan, bahkan dari VM, bahwa hanya akan ada satu Elvis.

```
// Enum singleton - the preferred approach

public enum Elvis {

        INSTANCE;

        private String[] favoriteSongs = { "Hound Dog", "Heartbreak Hotel" };

        public void printFavorites() {

                System.out.println(Arrays.toString(favoriteSongs));

        }

}
```

Simpulannya, kita seharusnya menggunakan tipe enum untuk mengontrol setiap jenis dari tipe instance. Jika ini tidak mungkin dan kita butuh sebuah kelas untuk menjadi serializable dan pengontrol instance, kita harus menyediakan method readResolve dan memastikan bahwa semua instance dari kelas tipe data primitif atau transient.

Item 78. Consider serialization proxies instead of serialized instances.

Sebagaimana telah dijelaskan sebelumnya, keputusan untuk meng-implement Serializable meningkatkan kemungkinan bug dan masalah keamanan kode program, karena itu menyebabkan instanced diciptakan dengan mekanisme yang spesial-diluar kebiasaan yang diletakkan dalam

konstruktor yang biasa. Ada, bagaimanapun teknik untuk mengurangi resiko-resiko itu. Teknik ini bernama *serialization proxy pattern.*

1. Rancang sebuah kelas bersarang yang static dan private dari kelas serializable yang secara singkat merepresentasikan logical state dari kelas yang ditumpangi itu.
2. Kelas bersarang ini dikenal dengan nama *serialization proxy.* seharusnya memiliki construktor tunggal, yang jenis parameternya adalah kelas yang ditumpangi/induknya.
3. Konstruktor ini hanya menggandakan data yang berasal dari parameternya. Tidak perlu melakukan testing mengenai konsistensi data dan defensif copy.
4. Dengan merancang, bentuk default serializable dengan serialization proxy adalah bentuk default serializable dari kelas induk. Kedua kelas induk maupun serialization proxy harus dideklarasikan sebagai serializable.

contohnya, dibawah ini adalah serialization proxy untuk kelas period sebelumnya,

```
// Serialization proxy for Period class

private static class SerializationProxy implements Serializable {

        private final Date start;

        private final Date end;

        SerializationProxy(Period p) {

                this.start = p.start;

                this.end = p.end;

        }

        private static final long serialVersionUID = 234098243823485285L; // Any number will do (Item 75)

}
```

selanjutnya, tambahkan method writeReplace dibawah pada kelas induknya. Method ini bisa meng-copy semua detail ke setiap kelas dengan menggunakan serialization proxy.

```
// writeReplace method for the serialization proxy pattern

private Object writeReplace() {

    return new SerializationProxy(this);

}
```

Kehadiran method ini menyebabkan sistem serialisasi mengutamakan instance dari SerializationProxy ketimbang instance dari kelas induknya. Dengan kata lain, method writeReplace mengkonversi instance dari kelas induknya dengan serialization proxy sebelum serialization.

Dengan method writeReplace, sistem serialisasi tidak akan men-generate instance dari kelas induk yang serializable, tapi penyerang mungkin memalsukan sebuah updaya untuk melanggar ketidakberagaman dari kelas itu. Untuk menjamin penyerang tidak berhasil tambahkan method readObject pada kelas induknya.

```
// readObject method for the serialization proxy pattern

private void readObject(ObjectInputStream stream) throws InvalidObjectException {

    throw new InvalidObjectException("Proxy required");

}
```

Terakhir sediakan method readResolve pada SerializationProxy class yang mengembalikan instance yang persis sama dengan kelas induknya. Kehadiran method ini membuat sistem langsung mengkonversi serialization proxy kembali pada instance kelas induknya saat proses deserialisasi.

Method readResolve menciptakan sebuah instance dari kelas induknya menggunakan hanya public API yang dimilinya, dan disitulah letak keindahan

dari pola ini. Dia membuang jauh-jauh pola yang diluar kebiasaan dari proses serialisasi, karena instance hasil deserialisasi diciptakan dari konstruktor yang sama, static factory dan method sebagaimana instance lainnya. Ini melepaskan kita dari keharusan untuk memastikan secara terpisah proses deserialisasi mematuhi ketunggalan/ketidakbergaman jenis kelas. Jika static factory dari kelas atau konstruktor menciptakan ketunggalan, dan instancenya me-maintain mereka, kita telah memastikan bahwa ketunggalan/ketidakberagaman akan di-maintain oleh serialisasi sama baiknya.

```
// readResolve method for Period.SerializationProxy

private Object readResolve() {

        return new Period(start, end); // Uses public constructor

}
```

1. tidak sama seperti sebelumnya, cara ini mengizinkan field dari Period bersifat final, yang dibutuhkan untuk membuat kelas Period benar-benar immutable.
2. kita juga tidak perlu membuat pengecekan validitas yang menjadi bagian dari proses deseriaisasi, kita juga tidak perlu khawatir tentang serangan yang membuat nilai jadi tidak konsisten.

Ada cara lain yang membuat serialization proxy lebih powerful dari defensif copy. Pola serialization proxy memperbolehkan instance dari deserialisasi untuk memiliki kelas yang berbeda dari instance yang telah di-serializable yang sebelumnya. Mungkin sebagian orang akan berpikir ini tidak berguna.

Membahas kembali mengenai EnumSet, kelas ini tidak memiliki public konstruktor, hanya static factory. Dari sudut pandang client, mereka mengembalikan instances dari EnumSet, tapi faktanya, mereka mengembalikan dua jenis subclass, tergantung dari ukuran enum type. Jika enum type memiliki 64 bit atau kurang, static factory mengembalikan RegularEnumSet, sebaliknya, mereka akan mengembalikan JumboEnumSet.

Sekarang bayangkan, apa yang terjadi jika kita melakukan serialisasi sebuah tipe enum set, enum type itu memiliki 64 bit elemen, kemudian ditambah lima element lebih banyak, kemudian lakukan deserialisasi pada enum itu. Jenisnya RegularEnumSet saat diserialisasi, tapi itu telah menjadi JumboEnumSet saat dilakukan deserialisasi. Faktanya apa yang sebenarnya terjadi, karena EnumSet menggunakan pola serialization proxy, kasusnya kita ingin tahu, di bawah adalah serialization proxy dari EnumSet.

```
// EnumSet's serialization proxy

private static class SerializationProxy <E extends Enum<E>> implements Serializable {

    // The element type of this enum set.

    private final Class<E> elementType;

    // The elements contained in this enum set.

    private final Enum[] elements;

    SerializationProxy(EnumSet<E> set) {

        elementType = set.elementType;

        elements = set.toArray(EMPTY_ENUM_ARRAY); // (Item 43)

    }

    private Object readResolve() {

        EnumSet<E> result = EnumSet.noneOf(elementType);

        for (Enum e : elements)

            result.add((E)e);

        return result;

    }

    private static final long serialVersionUID = 362491234563181265L;

}
```

Pola serialization proxy mempunyai dua batasan.

1. Tidak akan cocok pada kelas yang bisa di extend oleh client.
2. Tidak akan cocok dengan kelas yang objeknya mengandung siklus, jika kita berusaha untuk memanggil sebuah method dengan method readResolve dari serialization proxy, kita akan mendapatkan ClassCastException.

Akhirnya, tambahan power dan security oleh pola serialization proxy tidak gratis. Pada mesin penulis, kita memerlukan 14 persen cost tambahan pada memori daripada menggunakan defensif copy.

Simpulannya, gunakan pola serialization proxy kapanpun kita mengetahui bahwa kita harus menuliskan method readObject dan writeObject dalam kelas yang tidak akan bisa diextends oleh client. Pola ini mungkin merupakan cara termudah untuk membuat object serializable kokoh dan mendukung ketunggalan.